



**RED HAT®
TRAINING**



Comprehensive, hands-on training that solves real world problems

Automation with Ansible

Student Workbook (ROLE)

AUTOMATION WITH ANSIBLE

Ansible 2.3 DO407

Automation with Ansible

Edition 2 20170725 20170725

Authors: Chen Chang, Artur Glogowski, George Hacker, Razique Mahroua,
Adolfo Vazquez, Snehangshu Karmakar
Editor: Steven Bonneville

Copyright © 2017 Red Hat, Inc.

The contents of this course and all its modules and related materials, including handouts to audience members, are Copyright © 2017 Red Hat, Inc.

No part of this publication may be stored in a retrieval system, transmitted or reproduced in any way, including, but not limited to, photocopy, photograph, magnetic, electronic or other record, without the prior written permission of Red Hat, Inc.

This instructional program, including all material provided herein, is supplied without any guarantees from Red Hat, Inc. Red Hat, Inc. assumes no liability for damages or legal action arising from the use or misuse of contents or details contained herein.

If you believe Red Hat training materials are being used, copied, or otherwise improperly distributed please e-mail training@redhat.com or phone toll-free (USA) +1 (866) 626-2994 or +1 (919) 754-3700.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, Hibernate, Fedora, the Infinity Logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux® is the registered trademark of Linus Torvalds in the United States and other countries.

Java® is a registered trademark of Oracle and/or its affiliates.

XFS® is a registered trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

The OpenStack® Word Mark and OpenStack Logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Portions of this course were adapted from the Ansible Lightbulb project. The material from that project is available from <https://github.com/ansible/lightbulb> under the MIT License.

Document Conventions	ix
Notes and Warnings	ix
Introduction	xi
Automation with Ansible	xi
Orientation to the Classroom Environment	xii
Internationalization	xiv
1. Introducing Ansible	1
Overview of Ansible	2
Quiz: Ansible Architecture	7
Installing Ansible	9
Guided Exercise: Installing Ansible	11
Summary	13
2. Deploying Ansible	15
Building an Ansible Inventory	16
Quiz: Building an Ansible Inventory	20
Managing Ansible Configuration Files	22
Guided Exercise: Managing Ansible Configuration Files	29
Running Ad Hoc Commands	33
Guided Exercise: Running Ad Hoc Commands	38
Managing Dynamic Inventories	42
Guided Exercise: Managing Dynamic Inventories	46
Lab: Deploying Ansible	49
Summary	57
3. Implementing Playbooks	59
Writing and Running Playbooks	60
Guided Exercise: Writing and Running Playbooks	66
Implementing Multiple Plays	70
Guided Exercise: Implementing Multiple Plays	78
Lab: Implementing Playbooks	85
Summary	93
4. Managing Variables and Inclusions	95
Managing Variables	96
Guided Exercise: Managing Variables	106
Managing Facts	111
Guided Exercise: Managing Facts	118
Managing Inclusions	123
Guided Exercise: Managing Inclusions	132
Lab: Managing Variables and Inclusions	137
Summary	146
5. Implementing Task Control	147
Constructing Flow Control	148
Guided Exercise: Constructing Flow Control	155
Implementing Handlers	159
Guided Exercise: Implementing Handlers	162
Implementing Tags	167
Guided Exercise: Implementing Tags	174
Handling Errors	179
Guided Exercise: Handling Errors	183

Lab: Implementing Task Control	190
Summary	204
6. Implementing Jinja2 Templates	205
Describing Jinja2 Templates	206
Quiz: Describing Jinja2 Templates	209
Implementing Jinja2 Templates	211
Guided Exercise: Implementing Jinja2 Templates	213
Lab: Implementing Jinja2 Templates	216
Summary	221
7. Implementing Roles	223
Describing Role Structure	224
Quiz: Describing Role Structure	228
Creating Roles	230
Guided Exercise: Creating Roles	233
Deploying Roles with Ansible Galaxy	241
Guided Exercise: Deploying Roles with Ansible Galaxy	247
Lab: Implementing Roles	252
Summary	263
8. Optimizing Ansible	265
Selecting Hosts with Host Patterns	266
Guided Exercise: Selecting Hosts with Host Patterns	272
Configuring Delegation	277
Guided Exercise: Configuring Delegation	283
Configuring Parallelism	289
Guided Exercise: Configuring Parallelism	293
Lab: Optimizing Ansible	298
Summary	311
9. Implementing Ansible Vault	313
Configuring Ansible Vault	314
Guided Exercise: Configuring Ansible Vault	317
Executing with Ansible Vault	320
Guided Exercise: Executing with Ansible Vault	325
Lab: Implementing Ansible Vault	329
Summary	340
10. Troubleshooting Ansible	341
Troubleshooting Playbooks	342
Guided Exercise: Troubleshooting Playbooks	345
Troubleshooting Ansible Managed Hosts	351
Guided Exercise: Troubleshooting Ansible Managed Hosts	354
Lab: Troubleshooting Ansible	357
Summary	366
11. Implementing Ansible Tower	367
Introduction to Ansible Tower	368
Quiz: Introduction to Ansible Tower	372
Installing Ansible Tower	374
Guided Exercise: Installing Ansible Tower	379
Navigating the Ansible Tower Web Interface	381
Guided Exercise: Navigating the Ansible Tower Web Interface	390

Quiz: Implementing Ansible Tower	393
Summary	395
12. Implementing Ansible in a DevOps Environment	397
Provisioning Vagrant Machines	398
Guided Exercise: Provisioning Vagrant Machines	404
Deploying Vagrant in a DevOps Environment	407
Guided Exercise: Deploying Vagrant in a DevOps Environment	412
Lab: Implementing Ansible in a DevOps Environment	416
Summary	422
13. Comprehensive Review: Automation with Ansible	423
Comprehensive Review	424
Lab: Deploying Ansible	427
Lab: Creating Playbooks	432
Lab: Creating Roles and Using Dynamic Inventory	442
Lab: Optimizing Ansible	453
Lab: Deploying Ansible Tower and Executing Jobs	464
A. Ansible Lightbulb Licensing	471
Ansible Lightbulb License	472

Document Conventions

Notes and Warnings



Note

"Notes" are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.



Important

"Important" boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring a box labeled "Important" will not cause data loss, but may cause irritation and frustration.



Warning

"Warnings" should not be ignored. Ignoring warnings will most likely cause data loss.



References

"References" describe where to find external documentation relevant to a subject.

Introduction

Automation with Ansible

Automation with Ansible (DO407) is designed for system administrators who intend to use Ansible for automation, configuration, and management. Students will learn how to install and configure Ansible. Students will also create and run playbooks to configure systems, and learn to manage inventories. Students will manage encryption for Ansible with Ansible Vault, and deploy Ansible Tower and use it to manage systems. Students will use Ansible in a DevOps environment with Vagrant.

Objectives

- Automate system administration tasks on managed hosts with Ansible.
- Learn how to write Ansible playbooks to standardize task execution.
- Centrally manage playbooks through a web interface with Ansible Tower.

Audience

- System and cloud administrators who need to automate cloud provisioning, configuration management, application deployment, intra-service orchestration, and other IT needs.

Prerequisites

- Red Hat Certified System Administrator (RHCSA in Red Hat Enterprise Linux) certification or equivalent experience.

Orientation to the Classroom Environment

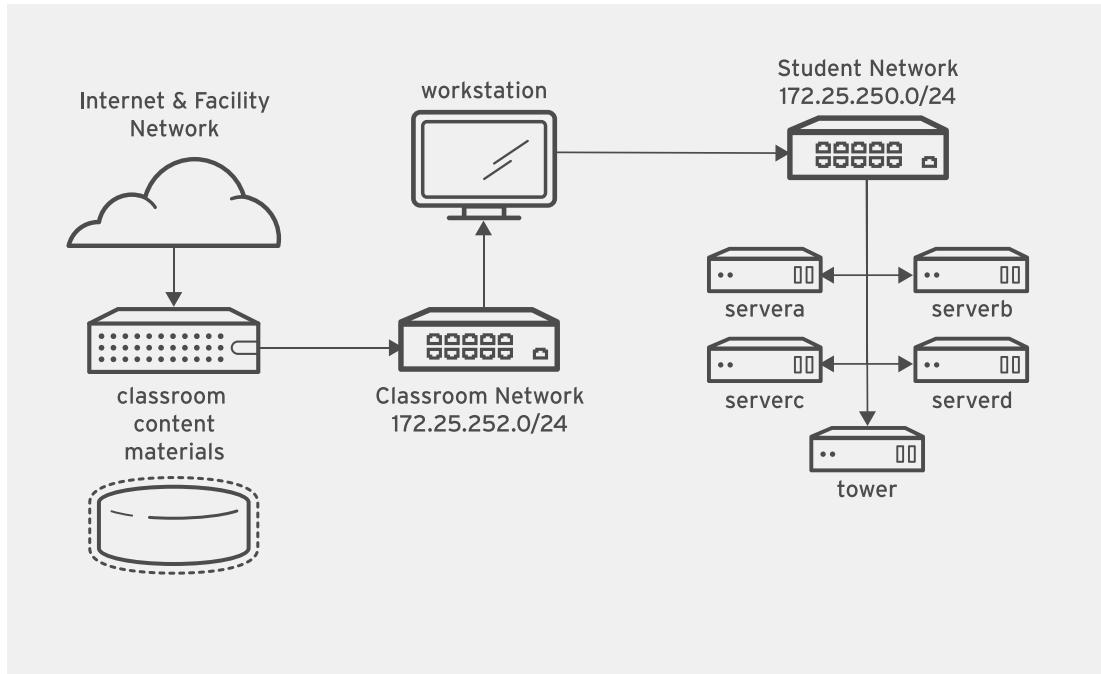


Figure 0.2: Classroom environment

In this course, the main computer system used for hands-on learning activities is **workstation**. Five other machines are also used by students for these activities: **servera**, **serverb**, **serverc**, **serverd**, and **tower**. All six of these systems are in the **lab.example.com** DNS domain.

All student computer systems have a standard user account, **student**, which has the password **student**. The **root** password on all student systems is **redhat**.

Classroom Machines

Machine name	IP addresses	Role
workstation.lab.example.com	172.25.250.254	Graphical workstation used to run most Ansible management commands
tower.lab.example.com	172.25.250.9	Host used for Ansible Tower and Vagrant
servera.lab.example.com	172.25.250.10	Host managed with Ansible
serverb.lab.example.com	172.25.250.11	Host managed with Ansible
serverc.lab.example.com	172.25.250.12	Host managed with Ansible
serverd.lab.example.com	172.25.250.13	Host managed with Ansible

The **workstation** machine also acts as a router between the network that connects the student machines and the classroom network. If **workstation** is down, other student machines will only be able to access systems on the student network.

Several systems in the classroom provide supporting services. Two servers, **content.example.com** and **materials.example.com**, act as sources for software and lab materials used in hands-on activities. Information on how to use these servers is provided in the instructions for those activities.

Controlling Your Station

The top of the console describes the state of your machine.

Machine States

State	Description
none	Your machine has not yet been started. When started, your machine will boot into a newly initialized state (the desk will have been reset).
starting	Your machine is in the process of booting.
running	Your machine is running and available (or, when booting, soon will be.)
stopping	Your machine is in the process of shutting down.
stopped	Your machine is completely shut down. Upon starting, your machine will boot into the same state as when it was shut down (the disk will have been preserved).
impaired	A network connection to your machine cannot be made. Typically this state is reached when a student has corrupted networking or firewall rules. If the condition persists after a machine reset, or is intermittent, please open a support case.

Depending on the state of your machine, a selection of the following actions will be available.

Machine Actions

Action	Description
Start Station	Start ("power on") the machine.
Stop Station	Stop ("power off") the machine, preserving the contents of its disk.
Reset Station	Stop ("power off") the machine, resetting the disk to its initial state. Caution: Any work generated on the disk will be lost.
Refresh	Refresh the page will re-probe the machine state.
Increase Timer	Adds 15 minutes to the timer for each click.

The Station Timer

Your Red Online Learning enrollment entitles you to a certain amount of computer time. In order to help you conserve your time, the machines have an associated timer, which is initialized to 60 minutes when your machine is started.

The timer operates as a "dead man's switch," which decrements while your machine is running. If the timer is winding down to 0, you can choose to increase the timer.

Internationalization

Language support

Red Hat Enterprise Linux 7 officially supports 22 languages: English, Assamese, Bengali, Chinese (Simplified), Chinese (Traditional), French, German, Gujarati, Hindi, Italian, Japanese, Kannada, Korean, Malayalam, Marathi, Odia, Portuguese (Brazilian), Punjabi, Russian, Spanish, Tamil, and Telugu.

Per-user language selection

Users may prefer to use a different language for their desktop environment than the system-wide default. They may also want to set their account to use a different keyboard layout or input method.

Language settings

In the GNOME desktop environment, the user may be prompted to set their preferred language and input method on first login. If not, then the easiest way for an individual user to adjust their preferred language and input method settings is to use the **Region & Language** application. Run the command **gnome-control-center region**, or from the top bar, select **(User) > Settings**. In the window that opens, select **Region & Language**. The user can click the **Language** box and select their preferred language from the list that appears. This will also update the **Formats** setting to the default for that language. The next time the user logs in, these changes will take full effect.

These settings affect the GNOME desktop environment and any applications, including **gnome-terminal**, started inside it. However, they do not apply to that account if accessed through an **ssh** login from a remote system or a local text console (such as **tty2**).



Note

A user can make their shell environment use the same **LANG** setting as their graphical environment, even when they log in through a text console or over **ssh**. One way to do this is to place code similar to the following in the user's **~/.bashrc** file. This example code will set the language used on a text login to match the one currently set for the user's GNOME desktop environment:

```
i=$(grep 'Language=' /var/lib/AccountService/users/${USER} \
| sed 's/Language=//')
if [ "$i" != "" ]; then
    export LANG=$i
fi
```

Japanese, Korean, Chinese, or other languages with a non-Latin character set may not display properly on local text consoles.

Individual commands can be made to use another language by setting the **LANG** variable on the command line:

```
[user@host ~]$ LANG=fr_FR.utf8 date
```

jeu. avril 24 17:55:01 CDT 2014

Subsequent commands will revert to using the system's default language for output. The **locale** command can be used to check the current value of **LANG** and other related environment variables.

Input method settings

GNOME 3 in Red Hat Enterprise Linux 7 automatically uses the **IBus** input method selection system, which makes it easy to change keyboard layouts and input methods quickly.

The **Region & Language** application can also be used to enable alternative input methods. In the **Region & Language** application's window, the **Input Sources** box shows what input methods are currently available. By default, **English (US)** may be the only available method. Highlight **English (US)** and click the **keyboard** icon to see the current keyboard layout.

To add another input method, click the **+** button at the bottom left of the **Input Sources** window. An **Add an Input Source** window will open. Select your language, and then your preferred input method or keyboard layout.

Once more than one input method is configured, the user can switch between them quickly by typing **Super+Space** (sometimes called **Windows+Space**). A *status indicator* will also appear in the GNOME top bar, which has two functions: It indicates which input method is active, and acts as a menu that can be used to switch between input methods or select advanced features of more complex input methods.

Some of the methods are marked with gears, which indicate that those methods have advanced configuration options and capabilities. For example, the Japanese **Japanese (Kana Kanji)** input method allows the user to pre-edit text in Latin and use **Down Arrow** and **Up Arrow** keys to select the correct characters to use.

US English speakers may find also this useful. For example, under **English (United States)** is the keyboard layout **English (international AltGr dead keys)**, which treats **AltGr** (or the right **Alt**) on a PC 104/105-key keyboard as a "secondary-shift" modifier key and dead key activation key for typing additional characters. There are also Dvorak and other alternative layouts available.



Note

Any Unicode character can be entered in the GNOME desktop environment if the user knows the character's Unicode code point, by typing **Ctrl+Shift+U**, followed by the code point. After **Ctrl+Shift+U** has been typed, an underlined **u** will be displayed to indicate that the system is waiting for Unicode code point entry.

For example, the lowercase Greek letter lambda has the code point U+03BB, and can be entered by typing **Ctrl+Shift+U**, then **03bb**, then **Enter**.

System-wide default language settings

The system's default language is set to US English, using the UTF-8 encoding of Unicode as its character set (**en_US.utf8**), but this can be changed during or after installation.

From the command line, **root** can change the system-wide locale settings with the **localectl** command. If **localectl** is run with no arguments, it will display the current system-wide locale settings.

To set the system-wide language, run the command **localectl set-locale LANG=locale**, where *locale* is the appropriate **\$LANG** from the "Language Codes Reference" table in this chapter. The change will take effect for users on their next login, and is stored in **/etc/locale.conf**.

```
[root@host ~]# localectl set-locale LANG=fr_FR.utf8
```

In GNOME, an administrative user can change this setting from **Region & Language** and clicking the **Login Screen** button at the upper-right corner of the window. Changing the **Language** of the login screen will also adjust the system-wide default language setting stored in the **/etc/locale.conf** configuration file.



Important

Local text consoles such as **tty2** are more limited in the fonts that they can display than **gnome-terminal** and **ssh** sessions. For example, Japanese, Korean, and Chinese characters may not display as expected on a local text console. For this reason, it may make sense to use English or another language with a Latin character set for the system's text console.

Likewise, local text consoles are more limited in the input methods they support, and this is managed separately from the graphical desktop environment. The available global input settings can be configured through **localectl** for both local text virtual consoles and the X11 graphical environment. See the **localectl(1)**, **kbd(4)**, and **vconsole.conf(5)** man pages for more information.

Language packs

When using non-English languages, you may want to install additional "language packs" to provide additional translations, dictionaries, and so forth. To view the list of available langpacks, run **yum langavailable**. To view the list of langpacks currently installed on the system, run **yum langlist**. To add an additional langpack to the system, run **yum langinstall code**, where *code* is the code in square brackets after the language name in the output of **yum langavailable**.



References

locale(7), **localectl(1)**, **kbd(4)**, **locale.conf(5)**, **vconsole.conf(5)**,
unicode(7), **utf-8(7)**, and **yum-langpacks(8)** man pages

Conversions between the names of the graphical desktop environment's X11 layouts and their names in **localectl** can be found in the file **/usr/share/X11/xkb/rules/base.lst**.

Language Codes Reference

Language Codes

Language	\$LANG value
English (US)	en_US.utf8
Assamese	as_IN.utf8
Bengali	bn_IN.utf8
Chinese (Simplified)	zh_CN.utf8
Chinese (Traditional)	zh_TW.utf8
French	fr_FR.utf8
German	de_DE.utf8
Gujarati	gu_IN.utf8
Hindi	hi_IN.utf8
Italian	it_IT.utf8
Japanese	ja_JP.utf8
Kannada	kn_IN.utf8
Korean	ko_KR.utf8
Malayalam	ml_IN.utf8
Marathi	mr_IN.utf8
Odia	or_IN.utf8
Portuguese (Brazilian)	pt_BR.utf8
Punjabi	pa_IN.utf8
Russian	ru_RU.utf8
Spanish	es_ES.utf8
Tamil	ta_IN.utf8
Telugu	te_IN.utf8



CHAPTER 1

INTRODUCING ANSIBLE

Overview	
Goal	Describe the terminology and architecture of Ansible.
Objectives	<ul style="list-style-type: none">• Describe Ansible concepts, reference architecture, and use cases.• Install Ansible.
Sections	<ul style="list-style-type: none">• Overview of Ansible (and Quiz)• Installing Ansible (and Guided Exercise)

Overview of Ansible

Objective

After completing this section, students should be able to describe Ansible concepts, architecture, and use cases.

What is Ansible?

Ansible is an open source automation platform. It's a *simple automation language* that can perfectly describe an IT application infrastructure in Ansible Playbooks. It's also an *automation engine* that runs Ansible Playbooks.

Ansible can manage powerful automation tasks, and can adapt to many different workflows and environments. At the same time, new users of Ansible can very quickly use it to become productive.

Ansible Is Simple

Ansible Playbooks provide human-readable automation. This means that your playbooks are automation tools that are also easy for humans to read, comprehend, and change. No special coding skills are required to write them. Playbooks execute tasks in order. The simplicity of playbook design makes them usable by every team. This allows people new to Ansible to get productive quickly.

Ansible Is Powerful

You can use Ansible to deploy applications, for configuration management, for workflow automation, and for network automation. Ansible can be used to orchestrate the entire application life cycle.

Ansible Is Agentless

Ansible is built around an agentless architecture. Typically, Ansible connects to the hosts it manages using OpenSSH or WinRM and runs tasks, often (but not always) by pushing out small programs called *Ansible modules* to those hosts. These programs are used to put the system in a specific desired state. Any modules pushed are removed when Ansible is finished with its tasks. It is possible to start using Ansible almost immediately, because no special agents need to be approved for use and then deployed to the managed hosts. Because there are no agents and no additional custom security infrastructure, Ansible is more efficient and more secure than other alternatives.

Ansible has a number of important strengths:

- *Cross platform support*: Ansible provides agentless support for Linux, Windows, UNIX, and network devices, in physical, virtual, cloud, and container environments.
- *Human-readable automation*: Ansible Playbooks, written as YAML text files, are easy to read and help ensure that everyone understands what they will do
- *Perfect description of applications*: Every change can be made by Ansible Playbooks, and every aspect of your application environment can be described and documented.
- *Easy to manage in version control*: Ansible Playbooks and projects are plain text. They can be treated like source code and placed in your existing version control system.

- *Support for dynamic inventories:* The list of machines that Ansible manages can be dynamically updated from external sources in order to capture the correct, current list of all managed servers all the time, regardless of infrastructure or location.
- *Orchestration that integrates easily with other systems:* HP SA, Puppet, Jenkins, Red Hat Satellite, and other systems that exist in your environment can be leveraged and integrated into your Ansible workflow.

Ansible: The Language of DevOps



Figure 1.1: Ansible across the application life cycle

Communication is the key to DevOps.

Ansible is the first automation language that can be read and written across IT. It is also the only automation engine that can automate the application life cycle and continuous delivery pipeline from start to finish.

Ansible Concepts and Architecture

There are two types of machines in the Ansible architecture: the *control nodes* and *managed hosts*. Ansible is installed and run from a control node, and this machine also has copies of your Ansible project files. A control node could be an administrator's laptop, a system shared by a number of administrators, or a server running Ansible Tower.

Managed hosts are listed in an *inventory*, which also organizes those systems into groups for easier collective management. The inventory can be defined in a static text file, or dynamically determined by scripts that get information from external sources.

Instead of writing complex scripts, Ansible users create high-level *plays* to ensure a host or group of hosts are in a particular state. A play performs a series of *tasks* on the host or hosts, in the order specified by the play. These plays are expressed in YAML format in a text file. A file that contains one or more plays is called a *playbook*.

Each task runs a *module*, a small piece of code (written in Python, PowerShell, or some other language), with specific arguments. Each module is essentially a tool in your toolkit. Ansible ships with hundreds of useful modules that can perform a wide variety of automation tasks. They can act on system files, install software, or make API calls.

When used in a task, a module generally ensures that some particular thing about the machine is in a particular state. For example, a task using one module may ensure that a file exists and has

particular permissions and contents, while a task using a different module may make certain that a particular file system is mounted. If the system is not in that state, the task should put it in that state. If the system is already in that state, it should do nothing. If a task fails, Ansible's default behavior is to abort the rest of the playbook for the hosts that had a failure. Tasks, plays, and playbooks should be *idempotent*. This means that you should be able to run a playbook on the same hosts multiple times safely, and when your systems are in the correct state the playbook should make no changes when run.

Ansible also uses *plug-ins*. Plug-ins are code that you can add to Ansible to extend it and adapt it to new uses and platforms.

The Ansible architecture is agentless. Typically, when an administrator runs an Ansible Playbook or an ad hoc command, the control node connects to the managed host using SSH (by default) or WinRM. This means that clients don't need to have an Ansible-specific agent installed on managed hosts, and don't need to permit special network traffic to some non-standard port.

Ansible Tower by Red Hat is an enterprise framework to help you control, secure, and manage your Ansible automation at scale. You can use it to control who has access to run playbooks on which hosts, share use of SSH credentials without allowing users to transfer or see their contents, log all of your Ansible jobs, and manage inventory, among many other things. It provides a web-based user interface and a RESTful API. It's not a core part of Ansible, but a separate product that helps you use Ansible more effectively with a team or at a large scale. We'll take a closer look at Ansible Tower later in this course.

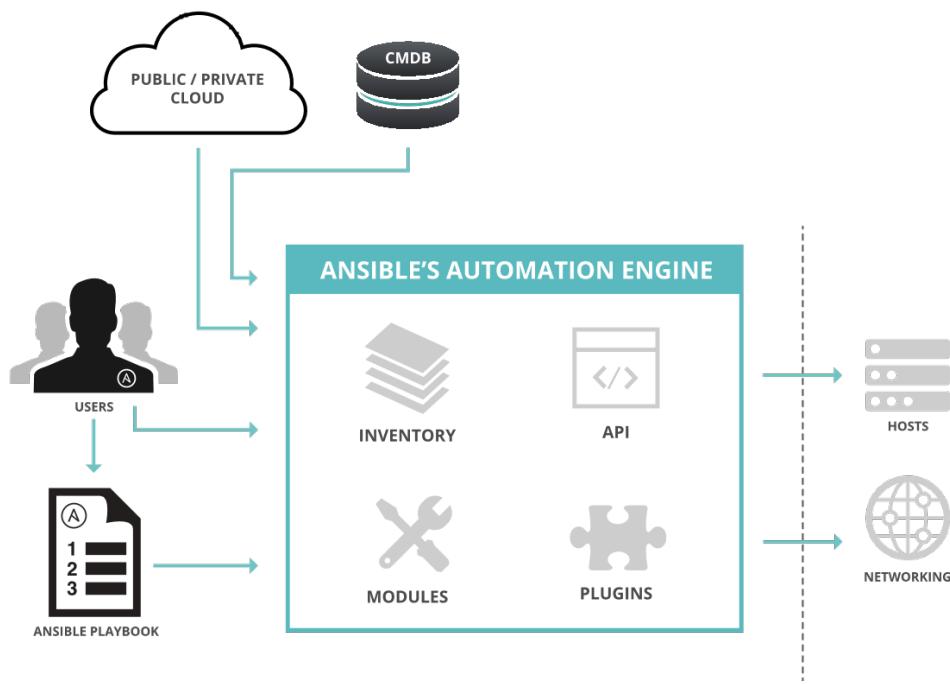


Figure 1.2: Ansible architecture

The Ansible Way

Complexity Kills Productivity

Simpler is better. Ansible is designed so that its tools are simple to use and automation is simple to write and read. You should take advantage of this to strive for simplification in how you create your automation.

Optimize For Readability

The Ansible automation language is built around simple, declarative, text-based files that are easy for humans to read. Written properly, Ansible Playbooks can clearly document your workflow automation.

Think Declaratively

Ansible is a desired-state engine. It approaches the problem of how to automate IT deployments by expressing them in terms of the state that you want your systems to be in. Ansible's goal is to put your systems into the desired state, only making changes that are necessary. Trying to treat Ansible like a scripting language is not the right approach.

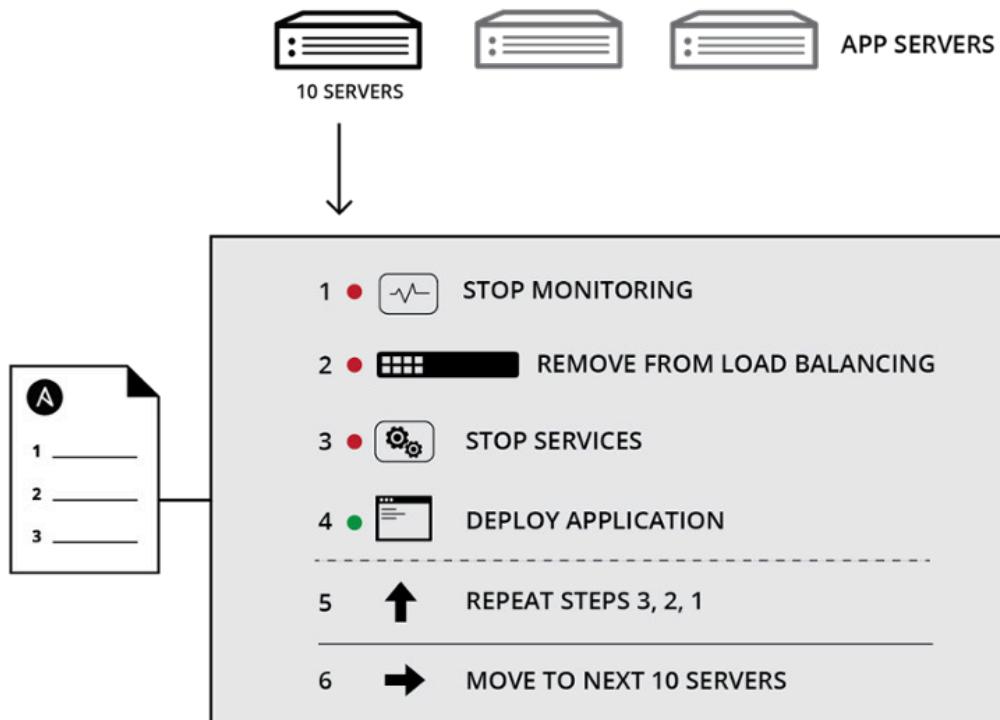


Figure 1.3: Ansible provides complete automation

Use Cases

Unlike some other tools, Ansible combines and unites orchestration with configuration management, provisioning, and application deployment in one easy-to-use platform.

Some use cases for Ansible include:

Configuration Management

Centralizing configuration file management and deployment is a common use case for Ansible, and it's how many power users are first introduced to the Ansible automation platform.

Application Deployment

When you define your application with Ansible, and manage the deployment with Ansible Tower, teams can effectively manage the entire application life cycle from development to production.

Provisioning

Applications have to be deployed or installed on systems. Ansible and Ansible Tower can help streamline the process of provisioning systems, whether you're PXE booting and kickstarting bare-metal servers or virtual machines, or creating virtual machines or cloud instances from templates.

Continuous Delivery

Creating a CI/CD pipeline requires coordination and buy-in from numerous teams. You can't do it without a simple automation platform that everyone in your organization can use. Ansible Playbooks keep your applications properly deployed (and managed) throughout their entire life cycle.

Security and Compliance

When your security policy is defined in Ansible, scanning and remediation of site-wide security policies can be integrated into other automated processes. Instead of being an afterthought, it is an integral part of everything that is deployed.

Orchestration

Configurations alone don't define your environment. You need to define how multiple configurations interact, and ensure the disparate pieces can be managed as a whole.

References

Ansible

<https://www.ansible.com>

How Ansible Works

<https://www.ansible.com/how-ansible-works>

Quiz: Ansible Architecture

Choose the correct answer to the following questions:

1. Which of the following terms best describes the Ansible architecture?
 - a. Agentless
 - b. Client/Server
 - c. Event-driven
 - d. Stateless

2. Which network protocol does Ansible use, by default, to communicate with managed nodes?
 - a. HTTP
 - b. HTTPS
 - c. SNMP
 - d. SSH

3. Which of the following files define the actions Ansible performs on managed nodes?
 - a. Host inventory
 - b. Manifest
 - c. Playbook
 - d. Script

4. What syntax is used to define Ansible playbooks?
 - a. Bash
 - b. Perl
 - c. Python
 - d. YAML

Solution

Choose the correct answer to the following questions:

1. Which of the following terms best describes the Ansible architecture?
 - a. **Agentless**
 - b. Client/Server
 - c. Event-driven
 - d. Stateless

2. Which network protocol does Ansible use, by default, to communicate with managed nodes?
 - a. HTTP
 - b. HTTPS
 - c. SNMP
 - d. **SSH**

3. Which of the following files define the actions Ansible performs on managed nodes?
 - a. Host inventory
 - b. Manifest
 - c. **Playbook**
 - d. Script

4. What syntax is used to define Ansible playbooks?
 - a. Bash
 - b. Perl
 - c. Python
 - d. **YAML**

Installing Ansible

Objectives

After completing this section, students should be able to:

- Install Ansible on the control node.
- Describe connection methods that are used to connect to managed hosts.

Control Nodes

Ansible is simple to install. The Ansible software only needs to be installed on the control node (or nodes) from which Ansible will be run. Hosts that are managed by Ansible do not need to have Ansible installed. This installation involves relatively few steps and has minimal requirements.

The control node should be a Linux or UNIX system. Microsoft Windows is not supported as a control node, although Windows systems can be managed hosts.

Python 2 (version 2.6 or later) needs to be installed on the control node. (Use of Python 3 with Ansible is still in technology preview and should not be used in production.) To see whether the appropriate version of Python is installed on a Red Hat Enterprise Linux system, use the **yum** command.

```
[root@controlnode ~]# yum list installed python
Loaded plugins: langpacks, search-disabled-repos
Installed Packages
python.x86_64      2.7.5-48.el7      installed
```

Official instructions on how to obtain, install, and get updates for Ansible for Red Hat Enterprise Linux, as well as for other operating systems and Linux distributions, are available on the Ansible website at <https://www.ansible.com/get-started>. This course is based on Ansible 2.3.

Ansible control nodes need to communicate with managed hosts over the network. By default, SSH is used, but other protocols might be needed if network devices or Microsoft Windows systems are being managed. On Red Hat Enterprise Linux control nodes, if you are managing Microsoft Windows systems, you also need to have version 0.2.2 or later of the *python2-winrm* RPM package installed (which provides the *pywinrm* Python package).

Managed Hosts

One of the benefits of Ansible is that managed hosts do not need to have a special agent installed. The Ansible control node connects to managed hosts using a standard network protocol to ensure that the systems are in the specified state.

Managed hosts might have some requirements depending on how the control node connects to them and what modules it will run on them.

Linux and UNIX managed hosts need to have Python 2 (version 2.4 or later) installed for most modules to work. For Red Hat Enterprise Linux, install the *python* package. If the version of Python installed on the managed host is earlier than Python 2.5, then the *python-simplejson* package must also be installed.

If SELinux is enabled on the managed hosts, you also need to install the *libselinux-python* package before using modules that are related to any copy, file, or template functions. (Note that if the other Python components are installed, you can use Ansible modules such as **yum** or **package** to ensure that this package is also installed.)

Some modules might have their own additional requirements. For example, the **dnf** module, which can be used to install packages on current Fedora systems, requires the *python-dnf* package.



Note

Some modules don't need Python at all. For example, arguments passed to the Ansible **raw** module are run directly through the configured remote shell instead of going through the module subsystem. This can be useful for managing devices that don't have Python available or can't have Python installed, or for bootstrapping Python onto a system that doesn't have it.

However, the **raw** module is difficult to use in a safely idempotent way. So if you can use a normal module instead, it's generally better to avoid using **raw** and the other command modules like it. We'll talk more about this later in the course.

Microsoft Windows-based Managed Hosts

Ansible includes a number of modules that are specifically designed for Microsoft Windows systems. These are listed in the *Windows Modules* [https://docs.ansible.com/ansible/list_of_windows_modules.html] section of the Ansible module index.

Most of the modules specifically designed for Microsoft Windows managed hosts require PowerShell 3.0 or higher on the managed host rather than Python. In addition, the managed hosts need to have PowerShell remoting configured.

This course uses Linux-based managed hosts in its examples, and does not go into great depth on the specific differences and adjustments needed when managing Microsoft Windows-based managed hosts. If you're interested, more information is available on the Ansible web site at https://docs.ansible.com/ansible/intro_windows.html.



References

ansible(1) man page

Installation – Ansible Documentation

http://docs.ansible.com/ansible/intro_installation.html

Windows Support

http://docs.ansible.com/ansible/intro_windows.html

Networking Support

http://docs.ansible.com/ansible/intro_networking.html

Guided Exercise: Installing Ansible

In this exercise, you will install Ansible on a control node and configure it for connections to a managed host.

Outcome

You should be able to install Ansible on a control node.

Before you begin

Log in as the **student** user on **workstation** and run **lab install setup**. This setup script ensures that the managed host, **servera**, is reachable on the network.

```
[student@workstation ~]$ lab install setup
```

Steps

- Verify that Python 2 is installed on **workstation**.

```
[student@workstation ~]$ yum list installed python
```

- Install Ansible on **workstation** so that it can be used as a control node.

```
[student@workstation ~]$ sudo yum install -y ansible
```

- Prepare to test the Ansible installation. Create a new directory, **/home/student/dep-install**. In that directory, create an Ansible inventory file named **inventory**. Follow the instructions below to edit that file so that it lists the managed host **servera.lab.example.com** as a member of the host group **dev**. (The exercise will show you how to do this for now, but the course will cover how static inventory files work in the next chapter.)

- 3.1. Create and change directory to the **/home/student/dep-install** directory.

```
[student@workstation ~]$ mkdir /home/student/dep-install  
[student@workstation ~]$ cd /home/student/dep-install
```

- 3.2. Use a text editor to create **/home/student/dep-install/inventory**, containing the following lines:

```
[dev]  
servera.lab.example.com
```

4. In the **/home/student/dep-install** directory, run the following **ansible** command to list all the managed hosts that are part of the **dev** group in the **/home/student/dep-install/inventory** inventory file:

```
[student@workstation dep-install]$ ansible dev -i inventory --list-hosts  
hosts (1):
```

```
servera.lab.example.com
```

You should see output similar to the preceding example, which lists **servera.lab.example.com** as the only host in the **dev** group.

5. Run **lab install grade** on **workstation** to grade your work.

```
[student@workstation ~]$ lab install grade
```

Summary

In this chapter, you learned:

- Ansible is an open source automation platform that can adapt to many different workflows and environments.
- Ansible can be used to manage many different types of systems, including servers running Linux, Microsoft Windows, or UNIX, and network devices.
- Ansible Playbooks are human-readable text files that describe the desired state of an IT infrastructure.
- Ansible is built around an agentless architecture in which Ansible is installed on a control node and clients do not need any special agent software.
- Ansible connects to managed hosts using standard network protocols such as SSH, and runs code or commands on the managed hosts to ensure that they are in the state specified by Ansible.



CHAPTER 2

DEPLOYING ANSIBLE

Overview	
Goal	Configure Ansible and run ad hoc commands.
Objectives	<ul style="list-style-type: none">• Describe Ansible inventory concepts and build a static inventory.• Manage Ansible configuration files.• Run Ansible ad hoc commands.• Manage dynamic inventory.
Sections	<ul style="list-style-type: none">• Building an Ansible Inventory (and Quiz)• Managing Ansible Configuration Files (and Guided Exercise)• Running Ad Hoc Commands (and Guided Exercise)• Managing Dynamic Inventory (and Guided Exercise)
Lab	<ul style="list-style-type: none">• Deploying Ansible

Building an Ansible Inventory

Objective

After completing this section, students should be able to describe Ansible inventory concepts and manage a static inventory.

The Inventory

An *inventory* defines a collection of hosts that Ansible will manage. These hosts can also be assigned to *groups*, which can be managed collectively. Groups can contain child groups, and hosts can be members of multiple groups. The inventory can also set variables that apply to the hosts and groups that it defines.

Host inventories can be defined in two different ways. A *static* host inventory can be defined by a text file. A *dynamic* host inventory can be generated by a script or other program as needed, using external information providers.

Static Inventory

A static inventory file is an INI-like text file that specifies the managed hosts that Ansible targets. In its simplest form, a static inventory is a list of host names or IP addresses of managed hosts, each on a single line:

```
web1.example.com
web2.example.com
db1.example.com
db2.example.com
192.0.2.42
```

Normally, however, you organize managed hosts into *host groups*. Host groups allow you to more effectively run Ansible against a collection of systems. In this case, each section starts with a host group name enclosed in square brackets ([]). This is followed by the host name or an IP address for each managed host in the group, each on a single line.

In the following example, the host inventory defines two host groups, **webservers** and **db-servers**.

```
[webservers]
web1.example.com
web2.example.com
192.0.2.42

[db-servers]
db1.example.com
db2.example.com
```

Hosts can be in multiple groups. In fact, recommended practice is to organize your hosts into multiple groups, possibly organized in different ways depending on the role of the host, its physical location, whether it's in production or not, and so on. This allows you to more easily apply Ansible plays to specific hosts.

```
[webservers]
```

```

web1.example.com
web2.example.com
192.168.3.7

[db-servers]
db1.example.com
db2.example.com

[east-datacenter]
web1.example.com
db1.example.com

[west-datacenter]
web2.example.com
db2.example.com

[production]
web1.example.com
web2.example.com
db1.example.com
db2.example.com

[development]
192.0.2.42

```



Important

Two host groups always exist:

- The **all** host group contains every host explicitly listed in the inventory.
- The **ungrouped** host group contains every host explicitly listed in the inventory that isn't a member of any other group.

Defining Nested Groups

Ansible host inventories can include groups of host groups. This is accomplished with the **:children** suffix. The following example creates a new group called **north-america**, which includes all of the hosts from the **usa** and **canada** groups.

```

[usa]
washington1.example.com
washington2.example.com

[canada]
ontario01.example.com
ontario02.example.com

[north-america:children]
canada
usa

```

A group can have both managed hosts and child groups as members. For example, in the previous inventory we could add a **[north-america]** section that has its own list of managed hosts. That list of hosts would be merged with the additional hosts the **north-america** group inherits from its child groups.

Simplifying Host Specifications with Ranges

Ansible host inventories can be simplified by specifying ranges in the host names or IP addresses. Numeric ranges can be specified, but alphabetic ranges are also supported. Ranges have the following syntax:

```
[START:END]
```

Ranges match all the values from *START* to *END*, inclusive. Consider the following examples:

- **192.168.[4:7].[0:255]** will match all IPv4 addresses in the 192.168.4.0/22 network (192.168.4.0 through 192.168.7.255).
- **server[01:20].example.com** will match all hosts named **server01.example.com** through **server20.example.com**.
- **[a:c].dns.example.com** will match hosts named **a.dns.example.com**, **b.dns.example.com**, and **c.dns.example.com**.
- **2001:db8::[a:f]** will match all IPv6 addresses from **2001:db8::a** through **2001:db8::f**.

If leading zeros are included in numeric ranges, they are used in the pattern. The second example above does not match **server1.example.com** but does match **server07.example.com**.

To illustrate this, the following example uses ranges to simplify the **usa** and **canada** group definitions from the earlier example:

```
[usa]
washington[1:2].example.com

[canada]
ontario[01:02].example.com
```

Testing the Inventory

When in doubt, test the machine's presence in the inventory with the **ansible** command:

```
[user@demo ~]$ ansible washington1.example.com --list-hosts
hosts (1):
  washington1.example.com
[user@demo ~]$ ansible washington01.example.com --list-hosts
[WARNING]: provided hosts list is empty, only localhost is available

hosts (0):
```

You can run the following command to list all hosts in a group:

```
[user@demo ~]$ ansible canada --list-hosts
hosts (2):
  ontario01.example.com
  ontario02.example.com
```



Important

If the inventory contains a host and a host group with the same name, the **ansible** command prints a warning and targets the host. The host group is ignored.

There are various ways to deal with this situation, the easiest being to ensure that host groups don't use the same names as hosts in the inventory.

Overriding the Location of the Inventory

The **/etc/ansible/hosts** file is considered the system's default static inventory file. However, normal practice is not to use that file but to define a different location for inventory files in your Ansible configuration file. This is covered in the next section.

The **ansible** and **ansible-playbook** commands you'll be using to run Ansible ad hoc commands and playbooks later in the course can also specify the location of an inventory file on the command line with the **--inventory PATHNAME** or **-i PATHNAME** option, where **PATHNAME** is the path to the desired inventory file.

Defining Variables in the Inventory

Values for variables used by playbooks can be specified in host inventory files. These variables only apply to specific hosts or host groups. Normally it is better to define these *inventory variables* in special directories and not directly in the inventory file. This topic is discussed in more depth elsewhere in the course.

Dynamic Inventory

Ansible inventory information can also be dynamically generated, using information provided by external databases. The open source community has written a number of dynamic inventory scripts that are available from the upstream Ansible project. If those scripts don't meet your needs, you can also write your own.

For example, a dynamic inventory program could contact your Red Hat Satellite server or Amazon EC2 account, and use information stored there to construct an Ansible inventory. Since the program does this when Ansible is run, it can populate the inventory with up-to-date information provided by the service as new hosts are added and old hosts are removed.

This topic is discussed in more detail later in this chapter.



References

Inventory: Ansible Documentation
http://docs.ansible.com/ansible/intro_inventory.html

Quiz: Building an Ansible Inventory

Choose the correct answers to the following questions:

1.

```
[linux-dev]
cchang.example.com
rlocke.example.com

[windows-dev]
wdinyes.example.com

[development:children]
linux-dev
windows-dev
```

Given the Ansible inventory in the above exhibit, which host group, or groups, includes the **rlocke.example.com** host?

- a. **linux-dev**
 - b. **windows-dev**
 - c. **development**
 - d. Both **linux-dev** and **development**
2. Which of the following expressions can be used in an Ansible inventory file to match hosts in the **10.1.0.0/16** address range?
- a. **10.1.0.0/16**
 - b. **10.1.[0:255].[0:255]**
 - c. **10.1.[0-255].[0-255]**
 - d. **10.1.***
3. In the inventory, a managed host
- a. must be in no more than one group other than **all**
 - b. may not be in a group that has child groups
 - c. may be in more than one group other than **all**
 - d. can not be listed by an IP address

Solution

Choose the correct answers to the following questions:

1.

```
[linux-dev]
cchang.example.com
rlocke.example.com

[windows-dev]
wdinyes.example.com

[development:children]
linux-dev
windows-dev
```

Given the Ansible inventory in the above exhibit, which host group, or groups, includes the **rlocke.example.com** host?

- a. **linux-dev**
 - b. **windows-dev**
 - c. **development**
 - d. **Both linux-dev and development**
2. Which of the following expressions can be used in an Ansible inventory file to match hosts in the **10.1.0.0/16** address range?
 - a. **10.1.0.0/16**
 - b. **10.1.[0:255].[0:255]**
 - c. **10.1.[0-255].[0-255]**
 - d. **10.1.***
 3. In the inventory, a managed host
 - a. must be in no more than one group other than **all**
 - b. may not be in a group that has child groups
 - c. **may be in more than one group other than all**
 - d. can not be listed by an IP address

Managing Ansible Configuration Files

Objectives

After completing this section, students should be able to:

- Describe Ansible configuration file locations.
- Describe Ansible configuration file syntax.
- Create Ansible configuration files and apply changes to default settings.

Configuring Ansible

The behavior of an Ansible installation can be customized by modifying settings in the Ansible configuration file. Ansible chooses its configuration file from one of several possible locations on the control node.

Using `/etc/ansible/ansible.cfg`

The `ansible` package provides a base configuration file located at `/etc/ansible/ansible.cfg`. This file is used if no other configuration file is found.

Using `~/.ansible.cfg`

Ansible looks for a `~/.ansible.cfg` in the user's home directory. This configuration is used instead of the `/etc/ansible/ansible.cfg` if it exists and if there is no `ansible.cfg` file in the current working directory.

Using `./ansible.cfg`

If an `ansible.cfg` file exists in the directory in which the `ansible` command is executed, it is used instead of the global file or the user's personal file. This allows administrators to create a directory structure where different environments or projects are stored in separate directories, with each directory containing a configuration file tailored with a unique set of settings.



Important

The recommended practice is to create an `ansible.cfg` file in a directory from which you run Ansible commands. This directory would also contain any files used by your Ansible project, such as an inventory and a playbook. This is the most common location used for the Ansible configuration file. It is unusual to use a `~/.ansible.cfg` or `/etc/ansible/ansible.cfg` file in practice.

Using `$ANSIBLE_CONFIG`

You can use different configuration files by placing them in different directories and then executing Ansible commands from the appropriate directory, but this method can be restrictive and hard to manage as the number of configuration files grows. A more flexible option is to define the location of the configuration file with the `$ANSIBLE_CONFIG` environment variable. When this variable is defined, Ansible uses the configuration file that the variable specifies instead of any of the previously mentioned configuration files.

Configuration File Precedence

The search order for a configuration file is the reverse of the preceding list. The first file located in the search order is the one from which Ansible will use configuration settings. Ansible will only use settings from this configuration file. Even if other files with lower precedence exist, their settings will be ignored and not combined with those in the selected configuration file.

Any file specified by the **\$ANSIBLE_CONFIG** environment variable will override all other configuration files. If that variable is not set, the directory in which the **ansible** command was run is checked for an **ansible.cfg** file next. If that file is not present, the user's home directory is checked for a **.ansible.cfg** file. The global **/etc/ansible/ansible.cfg** file is only used if no other configuration file is found.

Therefore, if you choose to create your own configuration file in favor of the global **/etc/ansible/ansible.cfg** configuration file, you need to duplicate all desired settings from that file to your own user-level configuration file. Settings not defined in the user-level configuration file remain set to the built-in defaults, even if they are set to some other value by the global configuration file.

Because of the multitude of locations in which Ansible configuration files can be placed, it can be confusing which configuration file is being used by Ansible, especially when multiple files exist on the control node. You can run the **ansible --version** command to clearly identify which version of Ansible is installed, and which configuration file is being used.

```
[student@controlnode ~]$ ansible --version
ansible 2.3.1.0
  config file = /etc/ansible/ansible.cfg
...output omitted...
```

Another way to display the active Ansible configuration file is to use the **-v** option when executing Ansible commands on the command line.

```
[student@controlnode ~]$ ansible servers --list-hosts -v
Using /etc/ansible/ansible.cfg as config file
...output omitted...
```

Managing Settings in the Configuration File

The Ansible configuration file consists of several sections, with each section containing settings defined as key-value pairs. Section titles are enclosed in square brackets. Settings are grouped under the following six sections in the default Ansible configuration file:

```
[student@controlnode ~]$ grep "\[" /etc/ansible/ansible.cfg
[defaults]
[privilege_escalation]
[paramiko_connection]
[ssh_connection]
[accelerate]
[selinux]
```

Most of the settings in the configuration file are grouped under the **[defaults]** section. The **[privilege_escalation]** section contains settings for defining how operations that require escalated privileges are executed on managed hosts. The **[paramiko_connection]**, **[ssh_connection]**, and **[accelerate]** sections contain settings for optimizing connections to managed hosts. The **[selinux]** section contains settings for defining how SELinux

interactions are configured. Although not included in the default global Ansible configuration file provided by the *ansible* package, a **[galaxy]** section is also available for defining parameters related to Ansible Galaxy, which is discussed in a later chapter.

Settings are customized by changing their values in the currently active configuration file. Changes take effect as soon as the file is saved. Some settings are predefined with default values within Ansible, and these values are valid even if their respective settings are commented out in the configuration file. To identify the default value of these predefined settings, consult the comments in the global **/etc/ansible/ansible.cfg** configuration file supplied by the *ansible* package. The **ansible** man page also provides information on the default values of these predefined settings.

Configuring Connections

Ansible needs to know how to communicate with its managed hosts. One of the most common reasons to change the configuration file is in order to control what methods and users Ansible will use to administer managed hosts. Some of the information needed includes:

- Where the inventory is that lists the managed hosts and host groups
- Which connection protocol to use to communicate with the managed hosts (by default, SSH), and whether a non-standard network port is needed to connect to the server
- Which remote user to use on the managed hosts; this could be **root** or it could be an unprivileged user
- If the remote user is unprivileged, Ansible needs to know whether it should try to escalate privileges to **root** and how to do it (for example, by using **sudo**)
- Whether or not to prompt for an SSH password or **sudo** password to log in or gain privileges

Inventory Location

In the **[defaults]** section, the **inventory** directive can point directly to a static inventory file, or to a directory that contains multiple static inventory files and/or dynamic inventory scripts:

```
[defaults]
inventory = ./inventory
```

Connection Settings

By default, Ansible connects to managed hosts using the SSH protocol. The most important parameters that control how Ansible connects to the managed hosts are set in the **[defaults]** section.

By default, Ansible attempts to connect to the managed host using the same user name as the local user running the Ansible commands. To specify a different remote user, set the **remote_user** parameter to that user name.

If the local user running Ansible has a private SSH key or keys configured that allow them to authenticate as the remote user on the managed hosts, Ansible automatically logs in. If that's not the case, you can configure Ansible to prompt the local user for the password used by the remote user by setting the directive **ask_pass = true**.

```
[defaults]
inventory = ./inventory
```

```
remote_user = root
ask_pass = true
```

Assuming that you're using a Linux control node and OpenSSH on your managed hosts, if you can log in to the remote user with a password then you can probably set up SSH key-based authentication, which would allow you to set **ask_pass = false**.

The first step is to make sure that the user on the control node has an SSH key pair configured in **~/.ssh**. You can run the **ssh-keygen** command to accomplish this.

For a single existing managed host, you can install your public key on the managed host and populate your local **~/.ssh/known_hosts** file with its host key using the **ssh-copy-id** command:

```
[student@controlnode ~]$ ssh-copy-id root@web1.example.com
The authenticity of host 'web1.example.com (192.168.122.181)' can't be established.
ECDSA key fingerprint is 70:9c:03:cd:de:ba:2f:11:98:fa:a0:b3:7c:40:86:4b.
Are you sure you want to continue connecting (yes/no)? yes
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter out any
that are already installed
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are prompted now
it is to install the new keys
root@web1.example.com's password:

Number of key(s) added: 1

Now try logging into the machine, with: "ssh 'root@web1.example.com'"
and check to make sure that only the key(s) you wanted were added.
```



Note

You can also use an Ansible Playbook to deploy your public key to the **remote_user** account on *all* managed hosts using the **authorized_key** module.

This course hasn't covered Ansible Playbooks in detail yet. For your future reference, a play that ensures that your public key is deployed to the managed hosts' **root** accounts might read:

```
- name: Public key is deployed to managed hosts for Ansible
hosts: all

tasks:
  - name: Ensure key is in root's ~/.ssh/authorized_hosts
    authorized_key:
      user: root
      state: present
      key: '{{ item }}'
    with_file:
      - ~/.ssh/id_rsa.pub
```

Because the managed host wouldn't have SSH key-based authentication configured yet, you would have to run the playbook using the **ansible-playbook** command with the **--ask-pass** option in order for the command to authenticate as the remote user.

Privilege Escalation

For security and auditing reasons, Ansible might need to connect to remote hosts as a non-privileged user before escalating privileges to get administrative access as **root**. This can be set up in the **[privilege_escalation]** section of the Ansible configuration file.

To enable privilege escalation by default, set the directive **become = true** in the configuration file. Even if this is set by default, there are various ways to override it when running ad hoc commands or Ansible Playbooks. (For example, there might be times when you want to run a task or play that does not escalate privileges.)

The **become_method** directive specifies how to escalate privileges. Several options are available, but the default is to use **sudo**. Likewise, the **become_user** directive specifies which user to escalate to, but the default is **root**.

If the **become_method** mechanism chosen requires the user to enter a password to escalate privileges, you can set the **become_ask_pass = true** directive in the configuration file.



Note

On Red Hat Enterprise Linux 7, the default configuration of **/etc/sudoers** grants all users in the **wheel** group the ability to use **sudo** to become **root** after entering their password.

One way to enable a user ("**someuser**" in the following example) to use **sudo** to become **root** without a password is to install a file with the appropriate directives into the **/etc/sudoers.d** directory (owned by **root**, with octal permissions 0400):

```
## password-less sudo for Ansible user
someuser ALL=(ALL) NOPASSWD:ALL
```

Think through the security implications of whatever approach you choose for privilege escalation. Different organizations and deployments might have different trade-offs to consider.

The following example **ansible.cfg** file assumes that you can connect to the managed hosts as **someuser** using SSH key-based authentication, and that **someuser** can use **sudo** to run commands as **root** without entering a password:

```
[defaults]
inventory = ./inventory
remote_user = someuser
ask_pass = false

[privilege_escalation]
become = true
become_method = sudo
become_user = root
become_ask_pass = false
```

The following table summarizes some of the most commonly modified directives in the Ansible configuration file.

Ansible Settings

Setting	Description
inventory	The location of the Ansible inventory.
remote_user	The remote user account used to establish connections to managed hosts.
ask_pass	Prompt for a password to use when connecting as the remote user.
become	Enable or disable privilege escalation for operations on managed hosts.
become_method	The privilege escalation method to use on managed hosts.
become_user	The user account to escalate privileges to on managed hosts.
become_ask_pass	Defines whether privilege escalation on managed hosts should prompt for a password.

Non-SSH Connections

The protocol used by Ansible to connect to managed hosts is set by default to **smart**, which determines the most efficient way to use SSH. This can be set to other values in a number of ways.

For example, there is one exception to the rule that SSH is used by default. If you do not have **localhost** in your inventory, Ansible sets up an *implicit localhost* entry to allow you to run ad hoc commands and playbooks that target **localhost**. This special inventory entry is not included in the **all** or **ungrouped** host groups. In addition, instead of using the **smart** SSH connection type, Ansible connects to it using the special **local** connection type by default.

```
[student@controlnode ~]$ ansible localhost --list-hosts
[WARNING]: provided hosts list is empty, only localhost is available

hosts (1):
  localhost
```

The **local** connection type ignores the **remote_user** setting and runs commands directly on the local system. If privilege escalation is being used, it runs **sudo** from the user account that ran the Ansible command, not **remote_user**. This can lead to confusion if the two users have different **sudo** privileges.

If you want to make sure that you connect to **localhost** using SSH like other managed hosts, one approach is to list it in your inventory. But this will include it in groups **all** and **ungrouped**, which you may not want to do.

Another approach is to change the protocol used to connect to **localhost**. The best way to do this is to set the **ansible_connection** host variable for **localhost**. To do this, in the directory from which you run Ansible commands, create a **host_vars** subdirectory. In that subdirectory, create a file named **localhost** that contains the line **ansible_connection: smart**. This ensures that the **smart** (SSH) connection protocol is used instead of **local** for **localhost**.

You can use this the other way around as well. If you have **127.0.0.1** listed in your inventory, by default you'll connect to it using **smart**. But you can create a **host_vars/127.0.0.1** file containing the line **ansible_connection: local** and it will use **local** instead.

Host variables will be covered in more detail later in the course.



Note

You can also use *group variables* to change the connection type for an entire host group. This can be done by placing files with the same name as the group in a **group_vars** directory, and ensuring that those files contain settings for the connection variables.

For example, you might want all your Microsoft Windows managed hosts to use the **winrm** protocol and port 5986 for connections. To configure this, you could put all of those managed hosts in group **windows**, and then create a file named **group_vars/windows** containing the following lines:

```
ansible_connection: winrm  
ansible_port: 5986
```

Configuration File Comments

There are two comment characters allowed by Ansible configuration files: the hash or number sign (#), and the semicolon (;).

The # character at the start of a line comments out the entire line. It must not be on the same line with a directive.

The ; character comments out everything to the right of it on the line. It can be on the same line as a directive, as long as that directive is to its left.



References

ansible(1), **ssh-keygen(1)**, and **ssh-copy-id(1)** man pages

Configuration file: Ansible Documentation
http://docs.ansible.com/ansible/intro_configuration.html

Guided Exercise: Managing Ansible Configuration Files

In this exercise, you will customize your Ansible environment.

Outcomes

You should be able to create a configuration file to configure your Ansible environment with persistent custom settings.

Before you begin

Log in as the **student** user on **workstation** and run **lab manage setup**. This setup script ensures that the managed host, **servera**, is reachable on the network.

```
[student@workstation ~]$ lab manage setup
```

Steps

1. Create the **/home/student/dep-manage** directory, which will contain the files for this exercise. Change to this newly created directory.

```
[student@workstation ~]$ mkdir /home/student/dep-manage  
[student@workstation ~]$ cd /home/student/dep-manage
```

2. In your **/home/student/dep-manage** directory, use a text editor to start editing a new file, **ansible.cfg**.

Create a **[defaults]** section in that file. In that section, add a line which uses the **inventory** directive to specify the **./inventory** file as the default inventory.

```
[defaults]  
inventory = ./inventory
```

Save your work and exit the text editor.

3. In the **/home/student/dep-manage** directory, use a text editor to start editing the new static inventory file, **inventory**.

The static inventory should contain three host groups:

- **myself** should contain the host **localhost**
- **intranetweb** should contain the host **servera.lab.example.com**
- **everyone** should contain the **myself** and **intranetweb** host groups

- 3.1. In **/home/student/dep-manage/inventory**, create the **myself** host group by adding the following lines.

```
[myself]  
localhost
```

- 3.2. In **/home/student/dep-manage/inventory**, create the **intranetweb** host group by adding the following lines.

```
[intranetweb]
servera.lab.example.com
```

- 3.3. In **/home/student/dep-manage/inventory**, create the **everyone** host group by adding the following lines.

```
[everyone:children]
myself
intranetweb
```



Note

Remember, you don't need to create a special group to be able to select all hosts in the inventory. You can just use the **all** host group. We're doing this to practice creating groups of groups.

- 3.4. Confirm that your final **inventory** file looks like this:

```
[myself]
localhost

[intranetweb]
servera.lab.example.com

[everyone:children]
myself
intranetweb
```

Save your work and exit the text editor.

4. Use **ansible** with the **--list-hosts** option to test the configuration of your inventory file's host groups. This will not actually connect to those hosts.

```
[student@workstation dep-manage]$ ansible myself --list-hosts
hosts (1):
  localhost
[student@workstation dep-manage]$ ansible intranetweb --list-hosts
hosts (1):
  servera.lab.example.com
[student@workstation dep-manage]$ ansible everyone --list-hosts
hosts (2):
  localhost
  servera.lab.example.com
```

5. Open the **/home/student/dep-manage/ansible.cfg** file in a text editor again. Add a **[privilege_escalation]** section to configure Ansible to automatically use **sudo** to switch from **student** to **root** when running tasks on the managed hosts. Ansible should also be configured to prompt you for the password that **student** uses for **sudo**.

-
- 5.1. Create the **privilege_escalation** section in the **/home/student/dep-manage/ansible.cfg** configuration file by adding the following entry.

```
[privilege_escalation]
```

- 5.2. Enable privilege escalation by setting the **become** directive to **true**.

```
become = true
```

- 5.3. Set the privilege escalation to use **sudo** by setting the **become_method** directive to **sudo**.

```
become_method = sudo
```

- 5.4. Set the privilege escalation user by setting the **become_user** directive to **root**.

```
become_user = root
```

- 5.5. Enable prompting for the privilege escalation password by setting the **become_ask_pass** directive to **true**.

```
become_ask_pass = true
```

- 5.6. Confirm that the complete **ansible.cfg** file looks like this:

```
[defaults]
inventory = ./inventory

[privilege_escalation]
become = true
become_method = sudo
become_user = root
become_ask_pass = true
```

Save your work and exit the text editor.

6. Run the **ansible --list-hosts** command again to verify that you are now prompted for the **sudo** password. Add the **-v** option to see the location of the current configuration file being used.

When prompted for the sudo password, enter **student**. (It won't be used for this dry run, however.)

```
[student@workstation dep-manage]$ ansible intranetweb --list-hosts -v
Using /home/student/dep-manage/ansible.cfg as config file
SUDO password: student
hosts (1):
servera.lab.example.com
```

7. Confirm that you can run Ansible tasks on the managed hosts. To do this, you will run your first *ad hoc command*. Ad hoc commands will be covered in more detail in the next section.

Run an **ansible** command targeting the **everyone** group, but replace **--list-hosts** with **-m ping**. That will run the **ping** module, which confirms that you can successfully run Ansible modules that use Python on the managed hosts.

This should work, assuming that the **student** user can log in to the managed hosts using SSH key-based authentication, and has **sudo** access on the managed hosts.

```
[student@workstation dep-manage]$ ansible everyone -m ping
SUDO password: student
localhost | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
servera.lab.example.com | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

Running Ad Hoc Commands

Objectives

After completing this section, students should be able to:

- Run ad hoc commands locally.
- Run ad hoc commands remotely.
- Discuss uses for ad hoc commands.

Performing Ad Hoc Commands with Ansible

An *ad hoc command* is a way to execute a single Ansible task quickly, one that you don't need to save to run again later. They're simple, one-line operations that can be run without writing a playbook.

They're useful for quick tests and changes. For example, you can use an ad hoc command to make sure a certain line exists in the **/etc/hosts** file on a group of servers. You could use another to efficiently restart a service on many different machines, or ensure that a particular software package is up-to-date. You could also use it to run an arbitrary command on one or more hosts to run a program or collect information.

Ad hoc commands are a very useful tool to quickly perform simple tasks with Ansible. They do have their limits, and in general you'll want to use Ansible Playbooks to realize the full power of Ansible. In many situations, however, ad hoc commands are exactly the tool you need to do something simple quickly.

Running Ad Hoc Commands

Use the **ansible** command to run ad hoc commands:

```
ansible host-pattern -m module [-a 'module arguments'] [-i inventory]
```

The *host-pattern* argument is used to specify the managed hosts on which the ad hoc command should be run. It could be a specific managed host or host group in the inventory. You've already seen this used in conjunction with the **--list-hosts** option, which shows you which hosts are matched by a particular host pattern. You've also already seen that you can use the **-i** option to specify a different inventory location to use than the default in the current Ansible configuration file.

The **-m** option takes as an argument the name of the *module* Ansible should run on the targeted hosts. Modules are small programs that are executed to implement your task. Some modules need no additional information, but others need additional arguments to specify the details of their operation. The **-a** option takes a list of those arguments as a quoted string.

One of the simplest ad hoc commands uses the **ping** module. This module doesn't do an ICMP ping, but checks to see if Python-based modules can be run on managed hosts. For example, the following ad hoc command determines whether all managed hosts in the inventory can run standard modules:

```
[student@controlnode ~]$ ansible all -m ping
servera.lab.example.com | SUCCESS => {
```

```
"changed": false,  
  "ping": "pong"  
}
```

Performing Tasks with Modules in Ad Hoc Commands

Modules are the tools that ad hoc commands use to accomplish tasks. Ansible provides hundreds of modules which do different things. You can usually find a tested, special-purpose module that does what you need as part of the standard installation.

The **ansible-doc -l** command lists all the modules that are installed on the system. You can then use **ansible-doc** to view the documentation of particular modules by name, and find information about what arguments the modules take as options. For example, the following command displays the documentation for the **ping** module, which has no options:

```
[student@controlnode ~]$ ansible-doc ping  
> PING      (/usr/lib/python2.7/site-packages/ansible/modules/system/ping.py)  
  
A trivial test module, this module always returns `pong' on successful contact. It  
does not make sense in playbooks, but it is useful from `/usr/bin/ansible' to verify  
the ability to login and that a usable python is configured. This is NOT ICMP ping,  
this is just a trivial test module.  
  
EXAMPLES:  
# Test we can logon to 'webservers' and execute python with json lib.  
ansible webservers -m ping  
  
MAINTAINERS: Ansible Core Team, Michael DeHaan  
  
METADATA:  
    Status: ['stableinterface']  
    Supported_by: core
```

Another place to learn about modules is in the online Ansible documentation at http://docs.ansible.com/ansible/modules_by_category.html.

The following modules might be immediately useful:

- File modules, such as **copy** (copy a local file to the managed host), **get_url** (download a file to the managed host), **synchronize** (to synchronize content like **rsync**), **file** (set permissions and other properties of a file), and **lineinfile** (make sure a certain line is or isn't in a file)
- Software package management modules, such as **yum**, **dnf**, **apt**, **pip**, **gem**, and so on
- System administration tools, such as **service**, to control daemons, and **user**, to add, remove, and configure users
- **uri**, which interacts with a web server and can test functionality or issue API requests

Most modules take arguments. The list of arguments available for a module can be found in the module's documentation. Ad hoc commands pass arguments to modules using the **-a** option. When no argument is needed, omit the **-a** option from the ad hoc command. If multiple arguments need to be specified, supply them as a quoted space-separated list.

For example, the following ad hoc command uses the **user** module to make sure that the **newbie** user exists and has UID 4000 on **servera.lab.example.com**:

```
[student@controlnode ~]$ ansible -m user -a 'name=newbie uid=4000 state=present' \
> servera.lab.example.com
servera.lab.example.com | SUCCESS => {
    "changed": true,
    "comment": "",
    "createhome": true,
    "group": 4000,
    "home": "/home/newbie",
    "name": "newbie",
    "shell": "/bin/bash",
    "state": "present",
    "system": false,
    "uid": 4000
}
```

Most modules are *idempotent*, which means that they can be run safely multiple times, and if the system is already in the correct state, they will do nothing. For example, we can run the previous ad hoc command again and we should see it report no changes:

```
[student@controlnode ~]$ ansible -m user -a 'name=newbie uid=4000 state=present' \
> servera.lab.example.com
servera.lab.example.com | SUCCESS => {
    "append": false,
    "changed": false,
    "comment": "",
    "group": 4000,
    "home": "/home/newbie",
    "move_home": false,
    "name": "newbie",
    "shell": "/bin/bash",
    "state": "present",
    "uid": 4000
}
```

Running Commands with the **command** Module

The **command** module allows administrators to execute arbitrary commands on the command line of managed hosts. The command to be executed is specified as an argument to the module using the **-a** option. For example, the following command executes the **hostname** command on the managed hosts referenced by the **mymanagedhosts** host pattern.

```
[student@controlnode ~]$ ansible mymanagedhosts -m command -a /usr/bin/hostname
host1.lab.example.com | SUCCESS | rc=0 >>
host1.lab.example.com
host2.lab.example.com | SUCCESS | rc=0 >>
host2.lab.example.com
```

The previous ad hoc command example returned two lines of output for each managed host. The first line is a status report, which shows the name of the managed host that the ad hoc operation was performed on, as well as the outcome of the operation. The second line is the output of the command executed remotely using the Ansible **command** module.

For better readability and parsing of ad hoc command output, administrators might find it useful to have a single line of output for each operation performed on a managed host. You can use the **-o** option to display the output of Ansible ad hoc commands in a single line format.

```
[student@controlnode ~]$ ansible mymanagedhosts -m command -a /usr/bin/hostname -o
host1.lab.example.com | SUCCESS | rc=0 >> (stdout) host1.lab.example.com
```

```
host2.lab.example.com | SUCCESS | rc=0 >> (stdout) host2.lab.example.com
```

The **command** module allows administrators to quickly execute remote commands on managed hosts. These commands are not processed by the shell on the managed hosts. As such, they cannot access shell environment variables or perform shell operations, such as redirection and piping.

For situations where commands require shell processing, administrators can use the **shell** module. Like the **command** module, you pass the commands to be executed as arguments to the module in the ad hoc command. Ansible then executes the command remotely on the managed hosts. Unlike the **command** module, the commands are processed through a shell on the managed hosts. Therefore, shell environment variables are accessible and shell operations such as redirection and piping are also available for use.

The following example illustrates the difference between the **command** and **shell** modules. If an attempt is made to execute the built-in Bash command **set** with these two modules, it will only succeed with the **shell** module.

```
[student@demo ~]$ ansible localhost -m command -a set
localhost | FAILED | rc=2 >>
[Errno 2] No such file or directory
[student@demo ~]$ ansible localhost -m shell -a set
localhost | SUCCESS | rc=0 >>
BASH=/bin/sh
BASHOPTS=cmdhist:extquote:force_fignore:hostcomplete:interact
ive_comments:progcomp:promptvars:sourcepath
BASH_ALIASES=()
...output omitted...
```

Both **command** and **shell** require a working Python installation on the managed host. A third module, **raw**, can run a command directly on the remote shell, bypassing the module subsystem. This is useful when managing systems that cannot have Python installed (for example, a network router). It can also be used to install Python on a host that doesn't have it yet.



Important

In most circumstances, it is a recommended practice that you avoid the **command**, **shell**, and **raw** "run command" modules.

Most other modules are idempotent and can perform change tracking automatically. They can test the state of systems and do nothing if those systems are already in the correct state. By contrast, it's much more complicated to use the "run command" modules in a way which will be idempotent. Depending on them might make it harder for you to be confident that re-running an ad hoc command or playbook won't cause an unexpected failure.

There are times when the "run command" modules are valuable tools and a good solution to a problem. If you do need to use them, it's probably best to try to use **command** first, resorting to **shell** or **raw** only if you need their special features.

Configuring Connections for Ad Hoc Commands

The settings for managed host connections and privilege escalation can be configured in the Ansible configuration file, and they can also be defined using options in ad hoc commands. When

defined using options in ad hoc commands, the settings take precedence over those configured in the Ansible configuration file. The following table shows the analogous command-line options for each configuration file setting.

Ansible Command-line Options

Setting	Command-line option
<code>inventory</code>	<code>-i</code>
<code>remote_user</code>	<code>-u</code>
<code>become</code>	<code>--become, -b</code>
<code>become_method</code>	<code>--become-method</code>
<code>become_user</code>	<code>--become-user</code>
<code>become_ask_pass</code>	<code>--ask-become-pass, -K</code>

Before configuring these settings using command-line options, their currently defined values can be determined by consulting the output of `ansible --help`.

```
[student@controlnode ~]$ ansible --help
...output omitted...
-b, --become           run operations with become (nopasswd implied)
--become-method=BECOME_METHOD
                      privilege escalation method to use (default=sudo),
                      valid choices: [ sudo | su | pbrun | pfexec | runas |
                      doas ]
--become-user=BECOME_USER
...output omitted...
-u REMOTE_USER, --user=REMOTE_USER
                      connect as this user (default=None)
```

References

`ansible(1)` man page

- Patterns: Ansible Documentation
http://docs.ansible.com/ansible/intro_patterns.html
- Introduction to Ad-Hoc Commands: Ansible Documentation
http://docs.ansible.com/ansible/intro_adhoc.html
- Module Index: Ansible Documentation
http://docs.ansible.com/ansible/modules_by_category
- command - Executes a command on a remote node: Ansible Documentation
http://docs.ansible.com/ansible/command_module.html
- shell - Execute commands in nodes: Ansible Documentation
http://docs.ansible.com/ansible/shell_module.html

Guided Exercise: Running Ad Hoc Commands

In this exercise, you will execute ad hoc commands on multiple managed hosts.

Outcomes

You should be able to execute commands on managed hosts on an ad hoc basis using privilege escalation.

We will execute ad hoc commands on **workstation** and **servera** using the **devops** user account. This account has the same **sudo** configuration on both **workstation** and **servera**.

Before you begin

Log in as the **student** user on **workstation** and run **lab adhoc setup**. This setup script ensures that the managed host, **servera**, is reachable on the network. It also creates and populates the **/home/student/dep-adhoc** working directory with materials used in this exercise.

```
[student@workstation ~]$ lab adhoc setup
```

Steps

1. Determine the **sudo** configuration for the **devops** account on both **workstation** and **servera**.
 - 1.1. Determine the **sudo** configuration for the **devops** account that was configured when **workstation** was built. Enter **student** if prompted for the password for the **student** account.

```
[student@workstation ~]$ sudo cat /etc/sudoers.d/devops
[sudo] password for student: student
devops ALL=(ALL) NOPASSWD: ALL
```

Note that the user has full **sudo** privileges but does not require password authentication.

- 1.2. Determine the **sudo** configuration for the **devops** account that was configured when **servera** was built.

```
[student@workstation ~]$ ssh devops@servera.lab.example.com
[devops@servera ~]$ sudo cat /etc/sudoers.d/devops
devops ALL=(ALL) NOPASSWD: ALL
[devops@servera ~]$ exit
```

Note that the user has full **sudo** privileges but does not require password authentication.

2. Change directory to **/home/student/dep-adhoc** and examine the contents of the **ansible.cfg** and **inventory** files.

```
[student@workstation ~]$ cd /home/student/dep-adhoc
[student@workstation dep-adhoc]$ cat ansible.cfg
[defaults]
inventory=inventory
```

```
[student@workstation dep-adhoc]$ cat inventory
[myself]
localhost

[intranetweb]
servera.lab.example.com

[everyone:children]
myself
intranetweb
```

The configuration file uses the directory's **inventory** file as the Ansible inventory. Note that we haven't configured Ansible to use privilege escalation yet.

3. Using the **ping** module, execute an ad hoc command to make sure all managed hosts in the **everyone** group can run Ansible modules using Python.

```
[student@workstation dep-adhoc]$ ansible everyone -m ping
servera.lab.example.com | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
localhost | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

4. Using the **command** module, execute an ad hoc command on **workstation** to identify the user account used by Ansible to perform operations on managed hosts. Use the **localhost** host pattern to connect to **workstation** for the ad hoc command execution. Because we are connecting locally, **workstation** is both the control node and managed host.

```
[student@workstation dep-adhoc]$ ansible localhost -m command -a 'id'
localhost | SUCCESS | rc=0 >>
uid=1000(student) gid=1000(student) groups=1000(student),10(wheel)
context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```

Notice that the ad hoc command was performed on the managed host as the **student** user.

5. Execute the previous ad hoc command on **workstation** but connect and perform the operation with the **devops** user account by using the **-u** option.

```
[student@workstation dep-adhoc]$ ansible localhost -m command -a 'id' -u devops
localhost | SUCCESS | rc=0 >>
uid=1001(devops) gid=1001(devops) groups=1001(devops)
context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```

Notice that the ad hoc command was performed on the managed host as the **devops** user.

6. Using the **command** module, execute an ad hoc command on **workstation** to display the contents of the **/etc/motd** file. Execute the command using the **devops** account.

```
[student@workstation dep-adhoc]$ ansible localhost -m command -a 'cat /etc/motd' -u devops
localhost | SUCCESS | rc=0 >>
```

Notice that the **/etc/motd** file is currently empty.

- Using the **copy** module, execute an ad hoc command on **workstation** to change the contents of the **/etc/motd** file so that it consists of the string "Managed by Ansible" followed by a newline. Execute the command using the **devops** account, but do not use the **--become** option to switch to **root**. The ad hoc command should fail due to lack of permissions.

```
[student@workstation dep-adhoc]$ ansible localhost -m copy -a 'content="Managed by
Ansible\n" dest=/etc/motd' -u devops
localhost | FAILED! => {
    "changed": false,
    "checksum": "4458b979ede3c332f8f2128385df4ba305e58c27",
    "failed": true,
    "msg": "Destination /etc not writable"
}
```

The ad hoc command failed because the **devops** user does not have permission to write the file.

- Try that again with privilege escalation. You could fix the settings in the **ansible.cfg** file, but for this example just use appropriate command line options of the **ansible** command.

Using the **copy** module, execute the previous command on **workstation** to change the contents of the **/etc/motd** file so that it consists of the string "Managed by Ansible" followed by a newline. Use the **devops** user to make the connection to the managed host, but perform the operation as the **root** user using the **--become** option.

```
[student@workstation dep-adhoc]$ ansible localhost -m copy -a 'content="Managed by
Ansible\n" dest=/etc/motd' -u devops --become
localhost | SUCCESS => {
    "changed": true,
    "checksum": "4458b979ede3c332f8f2128385df4ba305e58c27",
    "dest": "/etc/motd",
    "gid": 0,
    "group": "root",
    "md5sum": "65a4290ee5559756ad04e558b0e0c4e3",
    "mode": "0644",
    "owner": "root",
    "secontext": "system_u:object_r:etc_t:s0",
    "size": 19,
    "src": "/home/devops/.ansible/tmp/ansible-tmp-1463518320.68-167292050637471/
source",
    "state": "file",
    "uid": 0
}
```

Note that the command succeeded this time because the ad hoc command was executed with privilege escalation.

- Try that ad hoc command again on all hosts in the **everyone** host group. That will make sure that **/etc/motd** on both **workstation** and **servera** consist of the text "**Managed by Ansible**".

```
[student@workstation dep-adhoc]$ ansible everyone -m copy -a 'content="Managed by
Ansible\n" dest=/etc/motd' -u devops --become
```

```
localhost | SUCCESS => {
    "changed": false,
    "checksum": "4458b979ede3c332f8f2128385df4ba305e58c27",
    "dest": "/etc/motd",
    "gid": 0,
    "group": "root",
    "md5sum": "65a4290ee5559756ad04e558b0e0c4e3",
    "mode": "0644",
    "owner": "root",
    "secontext": "system_u:object_r:etc_t:s0",
    "size": 35,
    "src": "/home/vagrant/.ansible/tmp/ansible-tmp-1499301796.26-92607052136399/source",
    "state": "file",
    "uid": 0
}
servera.lab.example.com | SUCCESS => {
    "changed": true,
    "checksum": "4458b979ede3c332f8f2128385df4ba305e58c27",
    "dest": "/etc/motd",
    "gid": 0,
    "group": "root",
    "md5sum": "65a4290ee5559756ad04e558b0e0c4e3",
    "mode": "0644",
    "owner": "root",
    "secontext": "system_u:object_r:etc_t:s0",
    "size": 19,
    "src": "/home/devops/.ansible/tmp/ansible-tmp-1499301796.28-93111008249456/source",
    "state": "file",
    "uid": 0
}
```

You should see **SUCCESS** for both **localhost** and **servera**. However, **localhost** should report "**changed**: **false**" because the file is already in the correct state there. Likewise, **servera** should report "**changed**: **true**" because the ad hoc command updated the file to the correct state.

10. Using the **command** module, execute an ad hoc command to run **cat /etc/motd** to verify that the contents of the file have been successfully modified on both **workstation** and **servera**. Use the **everyone** host group and the **devops** user to specify and make the connection to the managed hosts. You do not need privilege escalation for this command to work.

```
[student@workstation dep-adhoc]$ ansible everyone -m command -a 'cat /etc/motd' -u devops
servera.lab.example.com | SUCCESS | rc=0 >>
Managed by Ansible

localhost | SUCCESS | rc=0 >>
Managed by Ansible
```

Managing Dynamic Inventories

Objective

After completing this section, students should be able to use an Ansible dynamic inventory to programmatically build an inventory from external data sources.

Generating Inventories Dynamically

The static inventory files you've worked with so far are easy to write, and are convenient for managing small infrastructures. When working with a large number of machines, however, or in an environment where machines come and go very quickly, it can be hard to keep the static inventory files up-to-date.

Most large IT environments have systems that keep track of which hosts are available and how they are organized. For example, there might be an external directory service maintained by a monitoring system such as Zabbix, or on FreeIPA or Active Directory servers. Installation servers such as Cobbler or management services such as Red Hat Satellite might track deployed bare-metal systems. In a similar way, cloud services such as Amazon Web Services EC2 or an OpenStack deployment, or virtual machine infrastructures based on VMWare or Red Hat Virtualization might be sources of information about the instances and virtual machines that come and go.

Ansible supports *dynamic inventory* scripts that retrieve current information from these types of sources whenever Ansible executes, allowing the inventory to be updated in real time. These scripts are executable programs that collect information from some external source and output the inventory in JSON format.

Dynamic inventory scripts are used just like static inventory text files. The location of the inventory is specified either directly in the current **ansible.cfg** file, or using the **-i** option. If the inventory file is executable, it is treated as a dynamic inventory program and Ansible attempts to run it to generate the inventory. If the file is not executable, it is treated as a static inventory.



Note

The inventory location can be configured in the **ansible.cfg** configuration file with the **inventory** parameter. By default, it is configured to be **/etc/ansible/hosts**.

Contributed Scripts

A number of existing dynamic inventory scripts have been contributed to the Ansible project by the open source community. They're not included in the *ansible* package or officially supported by Red Hat. They are available from the Ansible GitHub site at <https://github.com/ansible/ansible/tree/devel/contrib/inventory>.

Some of the data sources or platforms that are targeted by contributed dynamic inventory scripts include:

- Private cloud platforms, such as Red Hat OpenStack Platform.

- Public cloud platforms, such as Rackspace Cloud, Amazon Web Services EC2, or Google Compute Engine.
- Virtualization platforms, such as Red Hat Virtualization (oVirt) and VMware vSphere.
- Platform-as-a-Service solutions, such as OpenShift Container Platform.
- Life cycle management tools, such as Foreman (with Red Hat Satellite 6 or stand-alone) and Spacewalk (upstream of Red Hat Satellite 5).
- Hosting providers, such as Digital Ocean or Linode.

Each script might have its own dependencies and requirements in order to function. The contributed scripts are mostly written in Python, but that's not a requirement for dynamic inventory scripts.

Writing Dynamic Inventory Programs

If a dynamic inventory script does not exist for the directory system or infrastructure in use, it is possible to write a custom dynamic inventory program. It can be written in any programming language, and must return inventory information in JSON format when passed appropriate options.

If you want to write your own dynamic inventory script, more detailed information is available at *Developing Dynamic Inventory Sources* [http://docs.ansible.com/ansible/dev_guide/developing_inventory.html] in the *Ansible Developer Guide*. The following is a brief overview.

The script should start with an appropriate "shebang" line (for example, **`#!/usr/bin/python`**) and should be executable so that Ansible can run it.

When passed the **--list** option, the script must output a JSON-encoded hash/dictionary of all of the hosts and groups in the inventory to standard output.

In its simplest form, a group can be a list of managed hosts. In this example of the JSON-encoded output from an inventory script, **webservers** is a host group which has **web1.lab.example.com** and **web2.lab.example.com** as managed hosts in the group. The **databases** host group includes the **db1.lab.example.com** and **db2.lab.example.com** hosts as members.

```
[student@workstation ~]$ ./inventoryscript --list
{
    "webservers" : [ "web1.lab.example.com", "web2.lab.example.com" ],
    "databases"   : [ "db1.lab.example.com", "db2.lab.example.com" ]
}
```

Alternatively, each group's value can be a JSON hash/dictionary containing a list of each managed host, any child groups, and any group variables that might be set. The next example shows the JSON-encoded output for a more complex dynamic inventory. The **boston** group has two child groups (**backup** and **ipa**), three managed hosts of its own, and a group variable set (**example_host: false**).

```
{
    "webservers" : [
        "web1.demo.example.com",
        "web2.demo.example.com"
    ],
    "boston"      : {
        "ipa"     : {
            "hosts": [
                "ipa1.lab.example.com",
                "ipa2.lab.example.com"
            ],
            "vars": {
                "example_host": false
            }
        },
        "backup"  : {
            "hosts": [
                "boston1.lab.example.com",
                "boston2.lab.example.com",
                "boston3.lab.example.com"
            ],
            "vars": {
                "example_host": true
            }
        }
    }
}
```

```
"boston" : {
    "children" : [
        "backup",
        "ipa"
    ],
    "vars" : {
        "example_host" : false
    },
    "hosts" : [
        "server1.demo.example.com",
        "server2.demo.example.com",
        "server3.demo.example.com"
    ]
},
"backup" : [
    "server4.demo.example.com"
],
"ipa" : [
    "server5.demo.example.com"
],
"_meta" : {
    "hostvars" : {
        "server5.demo.example.com": {
            "ntpserver": "ntp.demo.example.com",
            "dnsserver": "dns.demo.example.com"
        }
    }
}
```

The script should also support the **--host managed-host** option. That option may print a JSON hash/dictionary consisting of variables which should be associated with that host. If it does not, it must print an empty JSON hash/dictionary.

```
[student@workstation ~]$ ./inventoryscript --host server5.demo.example.com
{
    "ntpserver" : "ntp.demo.example.com",
    "dnsserver" : "dns.demo.example.com"
}
```



Note

When called with the **--host hostname** option, the script must print a JSON hash/dictionary of the variables for the specified host (potentially an empty JSON hash or dictionary if there are no variables provided).

Optionally, if the **--list** option returns a top-level element called **_meta**, it is possible to return all host variables in one script call, which improves script performance. In that case, **--host** calls are not made.

See *Developing Dynamic Inventory Sources* [http://docs.ansible.com/ansible/developing_inventory.html] for more information.

Managing Multiple Inventories

Ansible supports the use of multiple inventories in the same run. If either the value passed to the **-i** option or the value of the **inventory** parameter in the configuration file is a directory, then

all inventory files included in the directory, either static or dynamic, are combined to determine the inventory. The executable files within that directory are used to retrieve dynamic inventories, and the other files are used as static inventories.

Inventory files should not depend on other inventory files or scripts in order to resolve. For example, if a static inventory file specifies that a particular group should be a child of another group, it also needs to have a placeholder entry for that group, even if all members of that group come from the dynamic inventory. Consider the **cloud-east** group in the following example:

```
[cloud-east]  
  
[servers]  
test.demo.example.com  
  
[servers:children]  
cloud-east
```

This ensures that no matter what the order is in which inventory files are parsed, all of them are internally consistent.



Note

The order in which inventory files are parsed is not specified by the documentation. Currently, when multiple inventory files exist, they seem to be parsed in alphabetical order. If one inventory source depends on information from another in order to make sense, whether it works or whether it throws an error may depend on the order in which they're loaded. Therefore, it's important to make sure that all files are self-consistent to avoid unexpected errors.

Ansible ignores files in an inventory directory if they end with certain suffixes. This can be controlled with the **inventory_ignore_extensions** directive in the Ansible configuration file being used. More information is available in the Ansible documentation.



References

- | Dynamic Inventory: Ansible Documentation
http://docs.ansible.com/ansible/intro_dynamic_inventory.html
- | Developing Dynamic Inventory Sources: Ansible Documentation
http://docs.ansible.com/ansible/developing_inventory.html

Guided Exercise: Managing Dynamic Inventories

In this exercise, you will install custom scripts that dynamically generate a list of inventory hosts.

Outcomes

You should be able to install and use existing dynamic inventory scripts.

Before you begin

Log in to **workstation** as **student** using **student** as the password.

On **workstation**, run the **lab deploy-dynamic setup** script. It checks if Ansible is installed on **workstation** and also creates a working directory for this exercise.

```
[student@workstation ~]$ lab deploy-dynamic setup
```

Steps

1. On workstation, change to the working directory for the exercise, **/home/student/dep-dynamic**.

```
[student@workstation ~]$ cd /home/student/dep-dynamic
```

2. Create an **ansible.cfg** Ansible configuration file in the working directory and populate it with the following entries so that the **inventory** directory is configured as the default inventory.

```
[defaults]
inventory = inventory
```

3. Create the **/home/student/dep-dynamic/inventory** directory.

```
[student@workstation dep-dynamic]$ mkdir inventory
```

4. From **http://materials.example.com/dynamic/**, download the **inventorya.py**, **inventoryw.py**, and **hosts** files to your **/home/student/dep-dynamic/inventory** directory. Both of the files ending in **.py** are scripts that generate dynamic inventories, and the third file is a static inventory.

- The **inventorya.py** script provides the **webservers** group, which includes the **servera.lab.example.com** host.
- The **inventoryw.py** script provides the **workstation.lab.example.com** host.
- The **hosts** static inventory file defines the **servers** group, which is a parent group of the **webservers** group.

```
[student@workstation dep-dynamic]$ wget http://materials.example.com/dynamic/
inventorya.py -O inventory/inventorya.py
```

```
[student@workstation dep-dynamic]$ wget http://materials.example.com/dynamic/inventoryw.py -O inventory/inventoryw.py  
[student@workstation dep-dynamic]$ wget http://materials.example.com/dynamic/hosts -O inventory/hosts
```

- Using the **ansible** command with the **inventorya.py** script as the inventory, list the managed hosts associated with the **webservers** group. It should raise an error relating to the permissions of **inventorya.py**.

```
[student@workstation dep-dynamic]$ ansible -i inventory/inventorya.py webservers --list-hosts  
ERROR! The file inventory/inventorya.py looks like it should be an executable inventory script, but is not marked executable. Perhaps you want to correct this with `chmod +x inventory/inventorya.py`?
```

- Check the current permissions for the **inventorya.py** script, and change them to 755.

```
[student@workstation dep-dynamic]$ ls -la inventory/inventorya.py  
-rw-rw-r--. 1 student student 0 Apr 29 14:20 inventory/inventorya.py  
[student@workstation dep-dynamic]$ chmod 755 inventory/inventorya.py
```

- Change the permissions for the **inventoryw.py** script to 755.

```
[student@workstation dep-dynamic]$ chmod 755 inventory/inventoryw.py
```

- Check the current output for the **inventorya.py** script using the **--list** parameter. The hosts associated with the **webservers** group are displayed.

```
[student@workstation dep-dynamic]$ inventory/inventorya.py --list  
{"webservers": {"hosts": ["servera.lab.example.com"], "vars": {} } }
```

- Check the current output for the **inventoryw.py** script using the **--list** parameter. The **workstation.lab.example.com** host is displayed.

```
[student@workstation dep-dynamic]$ inventory/inventoryw.py --list  
{"all": {"hosts": ["workstation.lab.example.com"], "vars": {} } }
```

- Check the **servers** group definition in the **/home/student/dep-dynamic/inventory/hosts** file. The **webservers** group defined in the dynamic inventory is configured as a child of the **servers** group.

```
[student@workstation dep-dynamic]$ cat inventory/hosts  
[servers:children]  
webservers
```

- Run the following command to verify the list of hosts in the **webservers** group. It raises an error about the **webservers** group being undefined.

```
[student@workstation dep-dynamic]$ ansible webservers --list-hosts  
ERROR! Attempted to read "/home/student/dep-dynamic/inventory/hosts" as YAML: Syntax Error while loading YAML.
```

```
The error appears to have been in '/home/student/dep-dynamic/inventory/hosts': line
1, column 9, but may
be elsewhere in the file depending on the exact syntax problem.
```

The offending line appears to be:

```
[servers:children]
  ^ here
```

```
Attempted to read "/home/student/dep-dynamic/inventory/hosts" as ini file: /home/
student/dep-dynamic/inventory/hosts:2: Section [servers:children] includes undefined
group: webservers
```

12. To make sure this problem doesn't happen, the static inventory should have a placeholder entry which defines an empty **webservers** host group. It's important for the static inventory to define any host group it references, because it's possible that it could dynamically disappear from the external source, which would cause this error.

Edit the **/home/student/dep-dynamic/inventory/hosts** file so it contains the following content:

```
[webservers]
[servers:children]
webservers
```



Important

If the dynamic inventory script that provides the host group is named so that it sorts before the static inventory referencing it, you might not see this error. However, if the host group ever disappears from the dynamic inventory, and you don't do this, the static inventory will be referencing a missing host group and the error will break the parsing of the inventory.

13. Rerun the following command to verify the list of hosts in the **webservers** group. It should work without any errors.

```
[student@workstation dep-dynamic]$ ansible webservers --list-hosts
hosts (1):
  servera.lab.example.com
```

Lab: Deploying Ansible

In this lab, you will configure an Ansible control node for connections to inventory hosts and use ad hoc commands to perform actions on managed hosts.

Outcomes

You should be able to configure a control node to run ad hoc commands on managed hosts.

You need to use Ansible to manage a number of hosts from **workstation.lab.example.com** as the **student** user. You will set up a project directory containing an **ansible.cfg** file with some specific defaults, and an **inventory** containing some inventory files and scripts.

You will then use ad hoc commands to ensure the **/etc/motd** file on all the machines in a particular host group consists of specified content.

Before you begin

Log in as the **student** user on **workstation** and run **lab deploy setup**. This setup script ensures that the managed hosts are reachable on the network.

```
[student@workstation ~]$ lab deploy setup
```

Steps

1. Verify that the **ansible** package is installed on the control node, and run the **ansible --version** command.
2. In the **student** user's home directory on **workstation**, **/home/student**, create a new directory named **dep-lab**. Change to that directory.
3. Create an **ansible.cfg** file in the **dep-lab** directory, which you should use to set the following Ansible defaults:
 - Connect to managed hosts as the **devops** user.
 - Use the **inventory** subdirectory to contain inventory files and scripts.
 - Disable privilege escalation by default. If privilege escalation is enabled from the command line, configure default settings to have Ansible use the **sudo** method to switch to the **root** user account. Ansible should not prompt for either the **devops** login password or the **sudo** password.

Your managed hosts have already been configured for you, with a **devops** user that **student** can log in as with SSH key-based authentication and that can run any command as **root** using **sudo** without a password.

4. Create the **/home/student/dep-lab/inventory** directory. Some inventory files have been provided for you to add to that directory:
 - Download **http://materials.example.com/dynamic/inventory** and save it as a static inventory file named **/home/student/dep-lab/inventory/inventory**. Modify the static inventory so that the host group **everyone** includes the child host group **internetweb**, which will be provided by the dynamic inventory script.

- Download <http://materials.example.com/dynamic/binventory.py> and save it as a dynamic inventory script named `/home/student/dep-lab/inventory/binventory.py`. Make sure its permissions allow it to be run by Ansible as a script.
5. Execute an ad hoc command targeting the **everyone** host group to verify that **devops** is the remote user and that privilege escalation is disabled by default.
6. Execute an ad hoc command, targeting your **everyone** host group, that uses the **copy** module to modify the contents of the `/etc/motd` file on all hosts in the group, based on the following instructions.

Use the **copy** module's **content** directive to ensure the `/etc/motd` file consists of the string **This server is managed by Ansible.\n** as a single line. (The `\n` used with the **content** directive causes the module to put a newline at the end of the string.)

You need to request privilege escalation from the command line to make this work with your current **ansible.cfg** defaults.

7. If you run the same ad hoc command again, you should see that the **copy** module detects that the files are already correct and does not change them. Look for the ad hoc command to report **SUCCESS** and the line "**changed": "false**" for each managed host.
8. To confirm this another way, run an ad hoc command that targets the **everyone** group, and which uses the **command** module to execute `cat /etc/motd`. The output from **ansible** should show the string "**This server is managed by Ansible.**" for all hosts. You don't need privilege escalation for this ad hoc command.
9. Run **lab deploy grade** on **workstation** to check your work.

```
[student@workstation dep-lab]$ lab deploy grade
```

Solution

In this lab, you will configure an Ansible control node for connections to inventory hosts and use ad hoc commands to perform actions on managed hosts.

Outcomes

You should be able to configure a control node to run ad hoc commands on managed hosts.

You need to use Ansible to manage a number of hosts from **workstation.lab.example.com** as the **student** user. You will set up a project directory containing an **ansible.cfg** file with some specific defaults, and an **inventory** containing some inventory files and scripts.

You will then use ad hoc commands to ensure the **/etc/motd** file on all the machines in a particular host group consists of specified content.

Before you begin

Log in as the **student** user on **workstation** and run **lab deploy setup**. This setup script ensures that the managed hosts are reachable on the network.

```
[student@workstation ~]$ lab deploy setup
```

Steps

1. Verify that the *ansible* package is installed on the control node, and run the **ansible --version** command.
 - 1.1. Verify that the *ansible* package is installed.

```
[student@workstation ~]$ yum list installed ansible
Installed Packages
ansible.noarch      2.3.1.0-1.el7      @ansible
```

- 1.2. Run the **ansible --version** command to confirm the version of Ansible that is installed.

```
[student@workstation ~]$ ansible --version
ansible 2.3.1.0
  config file = /etc/ansible/ansible.cfg
  configured module search path = Default w/o overrides
  python version = 2.7.5 (default, Aug  2 2016, 04:20:16) [GCC 4.8.5 20150623
(Red Hat 4.8.5-4)]
```

2. In the **student** user's home directory on **workstation**, **/home/student**, create a new directory named **dep-lab**. Change to that directory.

```
[student@workstation ~]$ mkdir /home/student/dep-lab
[student@workstation ~]$ cd /home/student/dep-lab
```

3. Create an **ansible.cfg** file in the **dep-lab** directory, which you should use to set the following Ansible defaults:
 - Connect to managed hosts as the **devops** user.
 - Use the **inventory** subdirectory to contain inventory files and scripts.

- Disable privilege escalation by default. If privilege escalation is enabled from the command line, configure default settings to have Ansible use the **sudo** method to switch to the **root** user account. Ansible should not prompt for either the **devops** login password or the **sudo** password.

Your managed hosts have already been configured for you, with a **devops** user that **student** can log in as with SSH key-based authentication and that can run any command as **root** using **sudo** without a password.

- 3.1. Use a text editor to create **/home/student/dep-lab/ansible.cfg**. Create a **[defaults]** section. Add a **remote_user** directive to have Ansible use the **devops** user when connecting to managed hosts. Also add an **inventory** directive to configure Ansible to use the **/home/student/dep-lab/inventory** directory as the default inventory.

```
[defaults]
remote_user = devops
inventory = inventory
```

- 3.2. In the **/home/student/dep-lab/ansible.cfg** file, create the **[privilege_escalation]** section and add the following entries to disable privilege escalation. Set the privilege escalation method to use the **root** account with **sudo** and without password authentication.

```
[privilege_escalation]
become = False
become_method = sudo
become_user = root
become_ask_pass = False
```

- 3.3. Confirm that the completed **ansible.cfg** file reads:

```
[defaults]
remote_user = devops
inventory = inventory

[privilege_escalation]
become = False
become_method = sudo
become_user = root
become_ask_pass = False
```

Save your work and exit the editor.

4. Create the **/home/student/dep-lab/inventory** directory. Some inventory files have been provided for you to add to that directory:
 - Download **http://materials.example.com/dynamic/inventory** and save it as a static inventory file named **/home/student/dep-lab/inventory/inventory**. Modify the static inventory so that the host group **everyone** includes the child host group **internetweb**, which will be provided by the dynamic inventory script.

- Download `http://materials.example.com/dynamic/binventory.py` and save it as a dynamic inventory script named `/home/student/dep-lab/inventory/binventory.py`. Make sure its permissions allow it to be run by Ansible as a script.

4.1. Create the `/home/student/dep-lab/inventory` directory.

```
[student@workstation dep-lab]$ mkdir inventory
```

4.2. Download the `http://materials.example.com/dynamic/inventory` file to the `/home/student/dep-lab/inventory` directory.

```
[student@workstation dep-lab]$ wget http://materials.example.com/dynamic/inventory -O inventory/inventory
```

4.3. Download the `http://materials.example.com/dynamic/binventory.py` script to the `/home/student/dep-lab/inventory` directory, and change its permission to 755.

```
[student@workstation dep-lab]$ wget http://materials.example.com/dynamic/binventory.py -O inventory/binventory.py
[student@workstation dep-lab]$ chmod 755 inventory/binventory.py
```

4.4. Configure the `internetweb` group as a child of the existing `everyone` group by adding the following entries to the `/home/student/dep-lab/inventory/inventory` file.

```
[internetweb]
[intranetweb]
servera.lab.example.com

[everyone:children]
intranetweb
internetweb
```

5. Execute an ad hoc command targeting the `everyone` host group to verify that `devops` is the remote user and that privilege escalation is disabled by default.

```
[student@workstation dep-lab]$ ansible everyone -m command -a 'id'
serverb.lab.example.com | SUCCESS | rc=0 >>
uid=1001(devops) gid=1001(devops) groups=1001(devops) context=unconfined_u:
unconfined_r:unconfined_t:s0-s0:c0.c1023

serverc.lab.example.com | SUCCESS | rc=0 >>
uid=1001(devops) gid=1001(devops) groups=1001(devops) context=unconfined_u:
unconfined_r:unconfined_t:s0-s0:c0.c1023

servera.lab.example.com | SUCCESS | rc=0 >>
uid=1001(devops) gid=1001(devops) groups=1001(devops) context=unconfined_u:
unconfined_r:unconfined_t:s0-s0:c0.c1023

serverd.lab.example.com | SUCCESS | rc=0 >>
uid=1001(devops) gid=1001(devops) groups=1001(devops) context=unconfined_u:
unconfined_r:unconfined_t:s0-s0:c0.c1023
```

Your results may be returned in a different order.

6. Execute an ad hoc command, targeting your **everyone** host group, that uses the **copy** module to modify the contents of the **/etc/motd** file on all hosts in the group, based on the following instructions.

Use the **copy** module's **content** directive to ensure the **/etc/motd** file consists of the string **This server is managed by Ansible.\n** as a single line. (The **\n** used with the **content** directive causes the module to put a newline at the end of the string.)

You need to request privilege escalation from the command line to make this work with your current **ansible.cfg** defaults.

```
[student@workstation dep-lab]$ ansible everyone -m copy \
> -a 'content="This server is managed by Ansible.\n" dest=/etc/motd' --become
servera.lab.example.com | SUCCESS => {
    "changed": true,
    "checksum": "93d304488245bb2769752b95e0180607effc69ad",
    "dest": "/etc/motd",
    "gid": 0,
    "group": "root",
    "md5sum": "af74293c7b2a783c4f87064374e9417a",
    "mode": "0644",
    "owner": "root",
    "secontext": "system_u:object_r:etc_t:s0",
    "size": 35,
    "src": "/home/devops/.ansible/tmp/ansible-tmp-1499275864.56-280761564717921/
source",
    "state": "file",
    "uid": 0
}
serverb.lab.example.com | SUCCESS => {
    "changed": true,
    "checksum": "93d304488245bb2769752b95e0180607effc69ad",
    "dest": "/etc/motd",
    "gid": 0,
    "group": "root",
    "md5sum": "af74293c7b2a783c4f87064374e9417a",
    "mode": "0644",
    "owner": "root",
    "secontext": "system_u:object_r:etc_t:s0",
    "size": 35,
    "src": "/home/devops/.ansible/tmp/ansible-tmp-1499275864.51-224886037138847/
source",
    "state": "file",
    "uid": 0
}
serverc.lab.example.com | SUCCESS => {
    "changed": true,
    "checksum": "93d304488245bb2769752b95e0180607effc69ad",
    "dest": "/etc/motd",
    "gid": 0,
    "group": "root",
    "md5sum": "af74293c7b2a783c4f87064374e9417a",
    "mode": "0644",
    "owner": "root",
    "secontext": "system_u:object_r:etc_t:s0",
    "size": 35,
    "src": "/home/devops/.ansible/tmp/ansible-tmp-1499275864.56-242019037094684/
source",
    "state": "file",
```

```

        "uid": 0
    }
serverd.lab.example.com | SUCCESS => {
    "changed": true,
    "checksum": "93d304488245bb2769752b95e0180607effc69ad",
    "dest": "/etc/motd",
    "gid": 0,
    "group": "root",
    "md5sum": "af74293c7b2a783c4f87064374e9417a",
    "mode": "0644",
    "owner": "root",
    "secontext": "system_u:object_r:etc_t:s0",
    "size": 35,
    "src": "/home/devops/.ansible/tmp/ansible-tmp-1499275864.58-48889952156589/
source",
    "state": "file",
    "uid": 0
}

```

7. If you run the same ad hoc command again, you should see that the **copy** module detects that the files are already correct and does not change them. Look for the ad hoc command to report **SUCCESS** and the line **"changed": false** for each managed host.

```

[student@workstation dep-lab]$ ansible everyone -m copy \
> -a 'content="This server is managed by Ansible.\n" dest=/etc/motd' --become
servera.lab.example.com | SUCCESS => {
    "changed": false,
    "checksum": "93d304488245bb2769752b95e0180607effc69ad",
    "dest": "/etc/motd",
    "gid": 0,
    "group": "root",
    "mode": "0644",
    "owner": "root",
    "path": "/etc/motd",
    "secontext": "system_u:object_r:etc_t:s0",
    "size": 35,
    "state": "file",
    "uid": 0
}
serverd.lab.example.com | SUCCESS => {
    "changed": false,
    "checksum": "93d304488245bb2769752b95e0180607effc69ad",
    "dest": "/etc/motd",
    "gid": 0,
    "group": "root",
    "mode": "0644",
    "owner": "root",
    "path": "/etc/motd",
    "secontext": "system_u:object_r:etc_t:s0",
    "size": 35,
    "state": "file",
    "uid": 0
}
serverc.lab.example.com | SUCCESS => {
    "changed": false,
    "checksum": "93d304488245bb2769752b95e0180607effc69ad",
    "dest": "/etc/motd",
    "gid": 0,
    "group": "root",
    "mode": "0644",
    "owner": "root",
    "path": "/etc/motd",

```

```
"secontext": "system_u:object_r:etc_t:s0",
  "size": 35,
  "state": "file",
  "uid": 0
}
serverb.lab.example.com | SUCCESS => {
  "changed": false,
  "checksum": "93d304488245bb2769752b95e0180607effc69ad",
  "dest": "/etc/motd",
  "gid": 0,
  "group": "root",
  "mode": "0644",
  "owner": "root",
  "path": "/etc/motd",
  "secontext": "system_u:object_r:etc_t:s0",
  "size": 35,
  "state": "file",
  "uid": 0
}
```

8. To confirm this another way, run an ad hoc command that targets the **everyone** group, and which uses the **command** module to execute **cat /etc/motd**. The output from **ansible** should show the string "**This server is managed by Ansible.**" for all hosts. You don't need privilege escalation for this ad hoc command.

```
[student@workstation dep-lab]$ ansible everyone -m command -a 'cat /etc/motd'
serverb.lab.example.com | SUCCESS | rc=0 >>
This server is managed by Ansible.

servera.lab.example.com | SUCCESS | rc=0 >>
This server is managed by Ansible.

serverd.lab.example.com | SUCCESS | rc=0 >>
This server is managed by Ansible.

serverc.lab.example.com | SUCCESS | rc=0 >>
This server is managed by Ansible.
```

9. Run **lab deploy grade** on **workstation** to check your work.

```
[student@workstation dep-lab]$ lab deploy grade
```

Summary

In this chapter, you learned:

- Any system on which Ansible is installed and which has access to the right configuration files and playbooks to manage remote systems (*managed hosts*) is called a *control node*.
- Managed hosts are defined in the *inventory*. Host patterns are used to reference managed hosts defined in an inventory.
- Inventories can be a static file or dynamically generated by a program from an external source, such as a directory service or cloud management system.
- The location of the inventory is controlled by the Ansible configuration file in use, but most frequently is kept with the playbook files.
- Ansible looks for its configuration file in a number of places in order of precedence. The first configuration file found is used; all others are ignored.
- The **ansible** command is used to perform *ad hoc commands* on managed hosts.
- Ad hoc commands determine the operation to perform through the use of *modules* and their arguments.
- Ad hoc commands requiring additional permissions can make use of Ansible's *privilege escalation* features.



CHAPTER 3

IMPLEMENTING PLAYBOOKS

Overview	
Goal	Write Ansible plays and execute a playbook.
Objectives	<ul style="list-style-type: none">• Write a basic Ansible Playbook and run it using the <code>ansible-playbook</code> command.• Write and run a more sophisticated Ansible Playbook using multiple plays and privilege escalation.
Sections	<ul style="list-style-type: none">• Writing and Running Playbooks (and Guided Exercise)• Implementing Multiple Plays (and Guided Exercise)
Lab	<ul style="list-style-type: none">• Implementing Playbooks

Writing and Running Playbooks

Objective

After completing this section, students should be able to write a basic Ansible Playbook and run it using the **ansible-playbook** command.

Ansible Playbooks and Ad Hoc Commands

Ad hoc commands can run a single, simple task against a set of targeted hosts as a one-time command. The real power of Ansible, however, is in learning how to use playbooks to run multiple, complex tasks against a set of targeted hosts in an easily repeatable manner.

A *play* is an ordered set of tasks which should be run against hosts selected from your inventory. A *playbook* is a text file that contains a list of one or more plays to run in order.

Plays allow you to change a lengthy, complex set of manual administrative tasks into an easily repeatable routine with predictable and successful outcomes. In a playbook, you can save the sequence of tasks in a play into a human-readable and immediately runnable form. The tasks themselves, because of the way in which they are written, document the steps needed to deploy your application or infrastructure.

Format of an Ansible Playbook

To help you understand the format of a playbook, we will review an ad hoc command that you saw in a previous chapter:

```
[student@controlnode ~]$ ansible -m user -a "name=newbie uid=4000 state=present" \
> servera.lab.example.com
```

This can be rewritten as a simple single-task play and saved in a playbook. The resulting playbook might appear as follows:

Example 3.1. A Simple Playbook

```
---
- name: Configure important user consistently
  hosts: servera.lab.example.com
  tasks:
    - name: newbie exists with UID 4000
      user:
        name: newbie
        uid: 4000
        state: present
```

A playbook is a text file written in YAML format, and is normally saved with the extension **.yml**. The playbook primarily uses indentation with space characters to indicate the structure of its data. YAML doesn't place strict requirements on how many spaces are used for the indentation, but there are two basic rules.

- Data elements at the same level in the hierarchy (such as items in the same list) must have the same indentation.

- Items that are children of another item must be indented more than their parents.

You can also add blank lines for readability.



Important

Only the space character can be used for indentation; tab characters are not allowed.

If you use the **Vim** text editor, you can apply some settings which might make it easier to edit your playbooks. For example, by adding the following line to your **\$HOME/.vimrc** file, when **vim** detects that you're editing a YAML file, it will perform a two space indentation when the **Tab** key is pressed, will autoindent subsequent lines, and will expand tabs into spaces.

```
autocmd FileType yaml setlocal ai ts=2 sw=2 et
```

The playbook begins with a line consisting of three dashes (---) as a start of document marker. It might also end with three dots (...) as an end of document marker, although in practice this is rarely used for playbooks.

In between those markers, the playbook is defined as a list of plays. An item in a YAML list starts with a single dash followed by a space. For example, a YAML list might appear as follows:

```
- apple
- orange
- grape
```

In *Example 3.1, “A Simple Playbook”*, the line after --- begins with a dash and starts the first (and only) play in the list of plays.

The play itself is a collection (an associative array or hash/dictionary) of key: value pairs. Keys in the same play should have the same indentation. The following example describes a YAML hash/dictionary with three keys. The first two keys have simple values. The third has a list of three items as a value.

```
name: just an example
hosts: webservers
tasks:
  - first
  - second
  - third
```

The original example play has three keys: **name**, **hosts**, and **tasks**. These keys all have the same indentation because they belong to the play.

The first line of the example play starts with a dash and a space (indicating the play is the first item of a list), and then the first key, the **name** attribute. The **name** associates an arbitrary string with the play as a label. This identifies what the play is for. The **name** key is optional, but is recommended because it helps to document your playbook. This is especially useful when a playbook contains multiple plays.

```
- name: Configure important user consistently
```

Chapter 3. Implementing Playbooks

The second key in the play is a **hosts** attribute, which specifies the hosts against which the play's tasks should be run. Like the argument for the **ansible** command, the **hosts** attribute takes a host pattern as a value, such as the names of managed hosts or groups in the inventory.

```
hosts: servera.lab.example.com
```

Finally, the last key in the play is the **tasks** attribute, whose value specifies a list of the tasks to run for this play. This example has a single task which runs the **user** module with specific arguments (to ensure user **newbie** exists and has UID 4000).

```
tasks:
  - name: newbie exists with UID 4000
    user:
      name: newbie
      uid: 4000
      state: present
```

The **tasks** attribute is the part of the play that actually lists, in order, the tasks to be run on the managed hosts. Each task in the list is itself a collection of key-value pairs.

In our example, the only task in the play has two keys:

- **name** is an optional label documenting the purpose of the task. It's a good idea to name all of your tasks to help document the purpose of each step of the automation process.
- **user** is the module to run for this task. Its arguments are passed as a collection of key-value pairs, which are children of the module (**name**, **uid**, and **state**).

The following is another example of a **tasks** attribute with multiple tasks, using the **service** module to ensure that several network services are enabled to start at boot:

```
tasks:
  - name: web server is enabled
    service:
      name: httpd
      enabled: true

  - name: NTP server is enabled
    service:
      name: chronyd
      enabled: true

  - name: Postfix is enabled
    service:
      name: postfix
      enabled: true
```



Important

The order in which the plays and tasks are listed in a playbook is important, because Ansible runs them in the same order.

The playbooks you've seen so far are basic examples, and you'll see more sophisticated examples of what you can do with plays and tasks as this course continues.

Running Playbooks

The **ansible-playbook** command is used to run playbooks. The command is executed on the control node and the name of the playbook to be run is passed as an argument:

```
[student@controlnode ~]$ ansible-playbook site.yml
```

When the playbook is executed, output is generated to show the play and tasks being executed. The output also reports the results of each task executed.

The following example shows the contents of a simple playbook, and then the result of running it.

```
[student@controlnode imp-playdemo]$ cat webserver.yml
---
- name: play to setup web server
  hosts: servera.lab.example.com
  tasks:
    - name: latest httpd version installed
      yum:
        name: httpd
        state: latest
...
[student@controlnode imp-playdemo]$ ansible-playbook webserver.yml

PLAY [play to setup web server] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [latest httpd version installed] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com : ok=2    changed=1    unreachable=0    failed=0
```

Note that the **name** set for each of your plays and tasks is displayed when the playbook is run. (The **Gathering Facts** task is a special task that the **setup** module usually runs automatically at the start of a play. This is covered later in the course.) For playbooks with multiple plays and tasks, setting **name** attributes makes it easier to monitor the progress of a playbook's execution.

You should also see that the **latest httpd version installed** task is "changed" for **servera.lab.example.com**. This means that the task changed something on that host to ensure its specification was met. In this case, it means that the *httpd* probably wasn't installed or wasn't the latest version.

In general, tasks in Ansible playbooks are idempotent, and it is safe to run the playbook multiple times. If the targeted managed hosts are already in the correct state, no changes should be made. For example, assume that the playbook from the previous example is run again:

```
[student@controlnode imp-playdemo]$ ansible-playbook webserver.yml

PLAY [play to setup web server] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [latest httpd version installed] ****
```

Chapter 3. Implementing Playbooks

```
ok: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com : ok=2    changed=0    unreachable=0    failed=0
```

This time, all tasks passed with status **ok** and no changes were reported.

Syntax Verification

Prior to executing a playbook, it is good practice to perform a verification to ensure that the syntax of its contents is correct. The **ansible-playbook** command offers a **--syntax-check** option which can be used to verify the syntax of a playbook file. The following example shows the successful syntax verification of a playbook.

```
[student@controlnode ~]$ ansible-playbook --syntax-check webserver.yml
playbook: webserver.yml
```

When syntax verification fails, a syntax error is reported. The output also includes the approximate location of the syntax issue in the playbook. The following example shows the failed syntax verification of a playbook where the space separator is missing after the **name** attribute for the play.

```
[student@controlnode ~]$ ansible-playbook --syntax-check webserver.yml
ERROR! Syntax Error while loading YAML.

The error appears to have been in '/home/student/webserver.yml': line 3, column 8, but
  may
be elsewhere in the file depending on the exact syntax problem.

The offending line appears to be:

  - name:play to setup web server
    hosts: servera.lab.example.com
      ^ here
```

Executing a Dry Run

Another helpful option is the **-C** option. This causes Ansible to report what changes would have occurred if the playbook were executed, but does not make any actual changes to managed hosts.

The following example shows the dry run of a playbook containing a single task for ensuring that the latest version of *httpd* package is installed on a managed host. Note that the dry run reports that the task would effect a change on the managed host.

```
[student@controlnode ~]$ ansible-playbook -C webserver.yml
PLAY [play to setup web server] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [latest httpd version installed] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
```

```
servera.lab.example.com : ok=2    changed=1    unreachable=0    failed=0
```



References

ansible-playbook(1) man page

Intro to Playbooks – Ansible Documentation

http://docs.ansible.com/ansible/playbooks_intro.html

Playbooks – Ansible Documentation

<http://docs.ansible.com/ansible/playbooks.html>

Check Mode ("Dry Run") – Ansible Documentation

http://docs.ansible.com/ansible/playbooks_checkmode.html

Guided Exercise: Writing and Running Playbooks

In this exercise, you will write and run your first Ansible playbook.

Outcomes

You should be able to write a playbook using basic YAML syntax and Ansible playbook structure, and successfully run it with the **ansible-playbook** command.

Before you begin

Log in as the **student** user on **workstation** and run **lab basic setup**. This setup script ensures that the managed hosts, **serverc.lab.example.com** and **serverd.lab.example.com**, are configured for the lab and are reachable on the network. It also ensures that the correct Ansible configuration file and inventory are installed on the control node.

```
[student@workstation ~]$ lab basic setup
```

A working directory, **/home/student/basic-playbook**, has been created on **workstation** for the Ansible project. The directory has already been populated with an **ansible.cfg** configuration file, and an **inventory** file, which defines a **web** group that includes both managed hosts listed above as members.

In this directory, use a text editor to create a playbook named **site.yml**. This playbook contains one play, which should target members of the **web** host group. The playbook should use tasks to ensure that the following conditions are met on the managed hosts:

1. The **httpd** package is present, using the **yum** module.
2. The local **files/index.html** file is copied to **/var/www/html/index.html** on each managed host, using the **copy** module.
3. The **httpd** service is started and enabled, using the **service** module.

You can use the **ansible-doc** command to help you understand the directives needed for each of the modules.

After the playbook is written, verify its syntax and then use **ansible-playbook** to run the playbook to implement the configuration.

Steps

1. To make all playbook exercises easier, if you use the **Vim** text editor you may want to use it to edit your **~/.vimrc** file (create it if necessary), to ensure it contains the following line:

```
autocmd FileType yaml setlocal ai ts=2 sw=2 et
```

This is optional, but it will set up the **vim** command so that the **Tab** key automatically indents using two space characters for YAML files. This may make it easier for you to edit Ansible playbooks.

2. Change directory to **/home/student/basic-playbook**.

```
[student@workstation ~]$ cd ~/basic-playbook
```

3. Use a text editor to create a new playbook, **/home/student/basic-playbook/site.yml**. Enter lines starting a play targeted at the hosts in the **web** host group.
 - 3.1. Create and open **~/basic-playbook/site.yml**. The first line of the file should be three dashes to indicate the start of the playbook.

```
---
```

- 3.2. The next line starts the play. It needs to start with a dash and a space before the first directive in the play. Name the play with an arbitrary string documenting what the play's purpose is, using the **name** directive.

```
- name: Install and start Apache HTTPD
```

- 3.3. Add a **hosts** directive which runs the play on hosts in the inventory's **web** host group. Make sure that the **hosts** directive is indented two spaces so it aligns with the **name** directive above.

The complete **site.yml** file should now appear as follows:

```
---
```

```
- name: Install and start Apache HTTPD
```

```
  hosts: web
```

4. Continue to edit the **/home/student/basic-playbook/site.yml** file, and add a **tasks** directive and the three tasks for your play that were specified in the instructions.

- 4.1. Add a **tasks** directive indented by two spaces (aligned with the **hosts** directive) to start the list of tasks. Your file should now appear as follows:

```
---
```

```
- name: Install and start Apache HTTPD
```

```
  hosts: web
```

```
    tasks:
```

- 4.2. Add the first task. Indent by four spaces, and start the task with a dash and a space, and then give the task a name, such as **httpd package is present**. The task should use the **yum** module. The module directives should be indented two more spaces; set the package **name** to **httpd** and the package **state** to **present**. The task should appear as follows:

```
    - name: httpd package is present
```

```
      yum:
```

```
        name: httpd
```

```
        state: present
```

- 4.3. Add the second task. Match the format of the previous task, and give the task a name, such as **correct index.html is present**. The task should use the **copy** module. The module directives should set **src** to **files/index.html** and **dest** to **/var/www/html/index.html**. The task should appear as follows:

```
- name: correct index.html is present
  copy:
    src: files/index.html
    dest: /var/www/html/index.html
```

- 4.4. Add the third task to start and enable the **httpd** service. Match the format of the previous two tasks, and give the new task a name, such as **httpd is started**. The task should use the **service** module. The module directives should set the **name** of the service to **httpd**, the **state** to **started**, and **enabled** to **true**. The task should appear as follows:

```
- name: httpd is started
  service:
    name: httpd
    state: started
    enabled: true
```

- 4.5. Your entire **site.yml** Ansible playbook should match the following example. Make sure that the indentation of your play's directives, the list of tasks, and each task's directives are all correct.

```
---
- name: Install and start Apache HTTPD
  hosts: web

  tasks:
    - name: httpd package is present
      yum:
        name: httpd
        state: present

    - name: correct index.html is present
      copy:
        src: files/index.html
        dest: /var/www/html/index.html

    - name: httpd is started
      service:
        name: httpd
        state: started
        enabled: true
```

Save the file and exit your text editor.

5. Before running your playbook, verify that its syntax is correct by running **ansible-playbook --syntax-check site.yml**. If it reports any errors, correct them before moving to the next step. You should see output similar to the following:

```
[student@workstation basic-playbook]$ ansible-playbook --syntax-check site.yml
```

```
playbook: site.yml
```

- Run your playbook. Read through the output generated to ensure that all tasks completed successfully.

```
[student@workstation basic-playbook]$ ansible-playbook site.yml

PLAY [Install and start Apache HTTPD] ****

TASK [Gathering Facts] ****
ok: [serverd.lab.example.com]
ok: [serverc.lab.example.com]

TASK [httpd package is present] ****
changed: [serverd.lab.example.com]
changed: [serverc.lab.example.com]

TASK [correct index.html is present] ****
changed: [serverd.lab.example.com]
changed: [serverc.lab.example.com]

TASK [httpd is started] ****
changed: [serverd.lab.example.com]
changed: [serverc.lab.example.com]

PLAY RECAP ****
serverc.lab.example.com    : ok=4      changed=3      unreachable=0      failed=0
serverd.lab.example.com    : ok=4      changed=3      unreachable=0      failed=0
```

- If all went well, you should be able to run the playbook a second time and see all tasks complete with no changes to the managed hosts.

```
[student@workstation basic-playbook]$ ansible-playbook site.yml

PLAY [Install and start Apache HTTPD] ****

TASK [Gathering Facts] ****
ok: [serverd.lab.example.com]
ok: [serverc.lab.example.com]

TASK [httpd package is present] ****
ok: [serverd.lab.example.com]
ok: [serverc.lab.example.com]

TASK [correct index.html is present] ****
ok: [serverc.lab.example.com]
ok: [serverd.lab.example.com]

TASK [httpd is started] ****
ok: [serverd.lab.example.com]
ok: [serverc.lab.example.com]

PLAY RECAP ****
serverc.lab.example.com    : ok=4      changed=0      unreachable=0      failed=0
serverd.lab.example.com    : ok=4      changed=0      unreachable=0      failed=0
```

Implementing Multiple Plays

Objectives

After completing this section, students should be able to:

- Write a playbook that uses multiple plays and per-play privilege escalation.
- Effectively use **ansible-doc** to discover module parameters to use in tasks and to evaluate how likely it is that a module's parameters will change.

Writing Multiple Plays

A playbook is a YAML file containing a list of one or more plays. Remember that a single play is an ordered list of tasks to execute against hosts selected from the inventory. Therefore, if a playbook contains multiple plays, each play may apply its tasks to a separate set of hosts.

This can be very useful when orchestrating a complex deployment which may involve different tasks on different hosts. A playbook can be written that runs one play against one set of hosts, and when that finishes runs another play against another set of hosts. (Of course, a second play could also run against the same set of hosts, if that made sense for some reason.)

Writing a playbook that contains multiple plays is very straightforward. Each play in the playbook is written as a top-level list item in the playbook. Each play is a list item containing the usual play directives.

The following example shows a simple playbook with two plays. The first play runs against **web.example.com**, and the second play runs against **database.example.com**.

```
---
# This is a simple playbook with two plays
- name: first play
  hosts: web.example.com
  tasks:
    - name: first task
      yum:
        name: httpd
        status: present
    - name: second task
      service:
        name: httpd
        enabled: true
- name: second play
  hosts: database.example.com
  tasks:
    - name: first task
      service:
        name: mariadb
        enabled: true
```

Remote Users and Privilege Escalation in Plays

Plays can use different remote users or privilege escalation settings for a play than what is specified by the defaults in the configuration file. These are set in the play itself at the same level as the **hosts** or **tasks** directives.

User Attributes

Tasks in playbooks are normally executed through a network connection to the managed hosts. As with ad hoc commands, the user account used for task execution depends on various parameters in the Ansible configuration file, **/etc/ansible/ansible.cfg**. The user that runs the tasks can be defined by the **remote_user** parameter. However, if privilege escalation is enabled, other parameters such **become_user** can also have an impact.

If the remote user defined in the Ansible configuration for task execution is not suitable, it can be overridden by using the **remote_user** attribute within a play.

```
remote_user: remoteuser
```

Privilege Escalation Attributes

Additional attributes are also available to define privilege escalation parameters from within a playbook. The **become** Boolean parameter can be used to enable or disable privilege escalation regardless of how it is defined in the Ansible configuration file. As usual, it can take **yes** or **true** to enable privilege escalation, or **no** or **false** to disable it.

```
become: true
```

If privilege escalation is enabled, the **become_method** attribute can be used to define the privilege escalation method to use during a specific play. The example below specifies that **sudo** be used for privilege escalation.

```
become_method: sudo
```

Additionally, with privilege escalation enabled, the **become_user** attribute can define the user account to use for privilege escalation within the context of a specific play.

```
become_user: privileged_user
```

The following example demonstrates the use of these directives in a play:

```
- name: /etc/hosts is up to date
  hosts: datacenter-west
  remote_user: automation
  become: yes

  tasks:
    - name: server.example.com in /etc/hosts
      lineinfile:
        path: /etc/hosts
        line: '192.0.2.42 server.example.com server'
        state: present
```

Finding Modules for Tasks

Module Documentation

The large number of modules packaged with Ansible provides administrators with many tools for common administrative tasks. Earlier in this course, we discussed the Ansible documentation website at <http://docs.ansible.com>. The module index on the website is an easy way to browse the list of modules shipped with Ansible. For example, modules for user and service management can be found under **Systems Modules** and modules for database administration can be found under **Database Modules**.

For each module, the Ansible documentation website provides a summary of its functions and instructions on how each specific function can be invoked with options to the module. The documentation also provides useful examples that show you how to use each module and how to set their parameters in a task.

You have already worked with the **ansible-doc** command to look up information about modules installed on the local system. As a review, to see a list of the modules available on a control node, run the **ansible-doc -l** command. This displays a list of module names and a synopsis of their function.

```
[student@workstation modules]$ ansible-doc -l
a10_server           Manage A10 Networks AX/SoftAX/Thunder/vThunder devices
a10_service_group    Manage A10 Networks devices' service groups
a10_virtual_server   Manage A10 Networks devices' virtual servers
acl                  Sets and retrieves file ACL information.
add_host              add a host (and alternatively a group) to the ansible-
playbook in-memory inventory
airbrake_deployment  Notify airbrake about app deployments
alternatives          Manages alternative programs for common commands
apache2_module        enables/disables a module of the Apache2 webserver
apk                  Manages apk packages
apt                  Manages apt-packages
...output omitted...
```

Detailed documentation on a specific module can be displayed by passing the module name to **ansible-doc**. Like the Ansible documentation website, the command provides a synopsis of the module's function, details of its various options, and examples. The following example shows the documentation displayed for the **yum** module.

```
[student@workstation modules]$ ansible-doc yum
> YUM

Installs, upgrade, removes, and lists packages and groups with the
`yum' package manager.

Options (= is mandatory):

- conf_file
  The remote yum configuration file to use for the transaction.
  [Default: None]

- disable_gpg_check
  Whether to disable the GPG checking of signatures of packages
  being installed. Has an effect only if state is `present' or
  `latest'. (Choices: yes, no) [Default: no]
```

```
...output omitted...

EXAMPLES:
- name: install the latest version of Apache
  yum: name=httpd state=latest

- name: remove the Apache package
  yum: name=httpd state=absent

...output omitted...
```

Module Maintenance

Ansible ships with a large number of modules that can be used for many tasks. The upstream community is very active, and these modules may be in different stages of development. The **ansible-doc** documentation for the module is expected to specify who provides maintenance for that module in the upstream Ansible community, and what its development status is. This is indicated in the **METADATA** section at the end of the output of **ansible-doc** for that module.

The **status** field records the development status of the module:

- **stableinterface**: the module's parameters are stable, and every effort will be made not to remove parameters or change their meaning.
- **preview**: the module is in technology preview, and might be unstable, its parameters might change, or it might require libraries or web services that are themselves subject to incompatible changes.
- **deprecated**: the module is deprecated, and will no longer be available in some future release.
- **removed**: the module has been removed from the release, but a stub exists for documentation purposes to help former users migrate to new modules.



Note

The **stableinterface** status only indicates that a module's interface is stable, it does not rate the module's code quality.

The **supported_by** field records who maintains the module in the upstream Ansible community. Possible values are:

- **core**: maintained by the "core" Ansible developers upstream, and always included with Ansible.
- **curated**: modules submitted and maintained by partners or companies in the community. Maintainers of these modules must watch for any issues reported or pull requests raised against the module. Upstream "core" developers review proposed changes to curated modules after the community maintainers have approved the changes. Core committers also ensure any issues with these modules due to changes in the Ansible engine are remediated. These modules are currently included with Ansible, but might be packaged separately at some point in the future.
- **community**: modules not supported by the core upstream developers or partners/companies, but maintained entirely by the general open source community. Modules in this category are still fully usable, but the response rate to issues is purely up to the community. These modules

are also currently included with Ansible, but will probably be packaged separately at some point in the future.

The upstream Ansible community has an issue tracker for Ansible and its integrated modules at <https://github.com/ansible/ansible/issues>.

Sometimes, a module doesn't exist for something you want to do. As an end user, you can also write your own private modules, or get modules from a third party. Ansible searches for custom modules in the location specified by the **ANSIBLE_LIBRARY** environment variable, or if that's not set, by a **library** directive in the current Ansible configuration file. Ansible also searches for custom modules in the **./library** directory relative to the playbook currently being run.

```
library = /usr/share/my_modules
```

Information on writing modules is beyond the scope of this course. Documentation on how to do this is available at http://docs.ansible.com/ansible/developing_modules.html.



Important

Use **ansible-doc** to find and learn how to use modules for your tasks.

When possible, try to avoid the **command**, **shell**, and **raw** modules in playbooks, even though they might seem simple to use. Because these take arbitrary commands, it is very easy to write non-idempotent playbooks with these modules.

For example, the following task using the **shell** module is not idempotent. Every time the play is run, it rewrites **/etc/resolv.conf** even if it already consists of the line "nameserver 192.0.2.1".

```
- name: Non-idempotent approach with shell module
  shell: echo "nameserver 192.0.2.1" > /etc/resolv.conf
```

There are several ways to write tasks using the **shell** module in an idempotent manner, and sometimes making those changes and using **shell** is the best approach. But a quicker solution may be to use **ansible-doc** to discover the **copy** module and use that to get the desired effect.

The following example does not rewrite the **/etc/resolv.conf** file if it already consists of the correct content:

```
- name: Idempotent approach with copy module
  copy:
    dest: /etc/resolv.conf
    content: "nameserver 192.0.2.1\n"
```

The **copy** module is special-purpose and can easily test to see if the state has already been met, and if so, it makes no changes. The **shell** module allows a lot of flexibility, but also requires more attention to ensure that it runs in an idempotent way.

Idempotent playbooks can be run repeatedly to ensure systems are in a particular state without disrupting those systems if they already are.

Playbook Syntax Variations

In the last part of this chapter, we'll look at some variations of YAML or Ansible Playbook syntax that you might encounter.

YAML Comments

Comments can also be used to aid readability. In YAML, everything to the right of the number or hash symbol (#) is a comment. If there is content to the left of the comment, precede the number symbol with a space.

```
# This is a YAML comment
```

```
some data # This is also a YAML comment
```

YAML Strings

Strings in YAML do not normally need to be put in quotation marks even if there are spaces contained in the string. If desired, strings can be enclosed in either double-quotes or single-quotes.

```
this is a string
```

```
'this is another string'
```

```
"this is yet another a string"
```

There are two ways to write multi-line strings. One way uses the vertical bar (|) character to denote that newline characters within the string are to be preserved.

```
include_newlines: |
    Example Company
    123 Main Street
    Atlanta, GA 30303
```

The other way to write multi-line strings uses the greater-than (>) character to indicate that newline characters are to be converted to spaces and that leading white spaces in the lines are to be removed. This method is often used to break long strings at space characters so that they can span multiple lines for better readability.

```
fold_newlines: >
    This is
    a very long,
    long, long, long
    sentence.
```

YAML Dictionaries

You've seen collections of key-value pairs written as an indented block, as follows:

```
name: svcrole
svcservice: httpd
svcport: 80
```

Chapter 3. Implementing Playbooks

Dictionaries can also be written in an inline block format enclosed in curly braces, as follows:

```
{name: svcrole, svcservice: httpd, svcport: 80}
```

In most cases the inline block format should be avoided because it is harder to read. However, there is at least one situation in which it is more commonly used. Later in the course, we will discuss *roles*. When a playbook is including a list of roles, it is more common to use this syntax in order to make it easier to distinguish roles being included in a play from the variables being passed to a role.

YAML Lists

You've also seen lists written with the normal single dash syntax:

```
hosts:
  - servera
  - serverb
  - serverc
```

Lists also have an inline format enclosed in square braces that looks like this:

```
hosts: [servera, serverb, serverc]
```

This should almost always be avoided because it's usually harder to read.

Obsolete key=value Playbook Shorthand

Some playbooks might use an older shorthand method to define tasks by putting the key-value pairs for the module on the same line as the module name. For example, you might see this syntax:

```
tasks:
  - name: shorthand form
    service: name=httpd enabled=true state=started
```

Normally you'd write the same task like this:

```
tasks:
  - name: normal form
    service:
      name: httpd
      enabled: true
      state: started
```

You should generally avoid the shorthand form and use the normal form.

The normal form has more lines, but it's easier to work with. The task's parameters are stacked vertically and easier to tell apart. Your eyes can run straight down the play with less left-to-right motion. Also, the normal syntax is native YAML, while the shorthand is not. Syntax highlighting tools in modern text editors can help you more effectively if you use the normal format than if you use the shorthand format.

However, you might run across this syntax in documentation and older playbooks from other people, and the syntax does still function.



References

ansible-playbook(1) and **ansible-doc(1)** man pages

Intro to Playbooks – Ansible Documentation

http://docs.ansible.com/ansible/playbooks_intro.html

Playbooks – Ansible Documentation

<http://docs.ansible.com/ansible/playbooks.html>

Developing Modules – Ansible Documentation

http://docs.ansible.com/ansible/developing_modules.html

Module Support – Ansible Documentation

http://docs.ansible.com/ansible/modules_support.html

Guided Exercise: Implementing Multiple Plays

In this exercise, you will write and use an Ansible playbook to perform administration tasks on a managed host.

Outcomes

You should be able to construct and execute a playbook to manage configuration and perform administration on a managed host.

Before you begin

Log in as the **student** user on **workstation** and run **lab playbook setup**. This setup script ensures that the managed host, **servera**, is reachable on the network. It also ensures that the correct Ansible configuration file and inventory are installed on the control node.

```
[student@workstation ~]$ lab playbook setup
```

A developer responsible for your company's intranet web site has asked you to write a playbook to help automate the setup of the server environment on **servera.lab.example.com**.

A working directory, **/home/student/imp-playbook**, has been created on **workstation** for the Ansible project. The directory has already been populated with an **ansible.cfg** configuration file and an **inventory** inventory file. The managed host, **servera.lab.example.com**, is already defined in this inventory file.

In this directory, create a playbook named **intranet.yml** which contains two plays. The first play requires privilege escalation and must perform the following tasks in the specified order:

1. Use the **yum** module to ensure that the latest versions of the **httpd** and **firewalld** packages are installed.
2. Ensure the **firewalld** service is enabled and started.
3. Ensure that **firewalld** is configured to allow connections to the **httpd** service.
4. Ensure that the **httpd** service is enabled and started.
5. Ensure that the managed host's **/var/www/html/index.html** file consists of the content "**Welcome to the example.com intranet!**".

The second play does not require privilege escalation and should run a single task using the **uri** module to confirm that the URL **http://servera.lab.example.com** returns an HTTP status code of 200.

Following recommended practices, plays and tasks should have names that document their purpose, but this is not required. The example solution names plays and tasks.

Don't forget that you can use the **ansible-doc** command to get help with finding and using the modules for your tasks.

After the playbook is written, verify its syntax and then execute the playbook to implement the configuration. Verify your work by executing **lab playbook grade**.

Steps

1. Change to the working directory, **/home/student/imp-playbook**.

```
[student@workstation ~] cd /home/student/imp-playbook
```

2. Create a new playbook, **/home/student/imp-playbook/intranet.yml**, and add the lines needed to start the first play. It should target the managed host **servera.lab.example.com** and enable privilege escalation.
- 2.1. Create and open a new playbook, **/home/student/imp-playbook/intranet.yml**, and add a line consisting of three dashes to the beginning of the file to indicate the start of the YAML file.

```
---
```

- 2.2. Add the following line to the **/home/student/imp-playbook/intranet.yml** file to denote the start of a play with a name of **Enable intranet services**.

```
- name: Enable intranet services
```

- 2.3. Add the following line to the **/home/student/imp-playbook/intranet.yml** file to indicate that the play applies to the **servera** managed host. Be sure to indent the line with two spaces (aligning with the **name** directive above it) to indicate that it is part of the first play.

```
hosts: servera.lab.example.com
```

- 2.4. Add the following line to the **/home/student/imp-playbook/intranet.yml** file to enable privilege escalation. Be sure to indent the line with two spaces (aligning with the directives above it) to indicate it is part of the first play.

```
become: yes
```

3. Add the following line to the **/home/student/imp-playbook/intranet.yml** file to define the beginning of the **tasks** list. Indent the line with two spaces (aligning with the directives above it) to indicate that it is part of the first play.

```
tasks:
```

4. As the first task in the first play, define a task that makes sure that the *httpd* and *firewalld* packages are up to date.
 - 4.1. Under the **tasks** directive in the first play, add the following lines to the **/home/student/imp-playbook/intranet.yml** file. This creates the task that ensures that the latest versions of the *httpd* and *firewalld* packages are installed.

Be sure to indent the first line of the task with four spaces, a dash, and a space. This indicates that the task is an item in the **tasks** list for the first play.

The first line provides a descriptive name for the task. The second line is indented with six spaces and calls the **yum** module. The next line is indented eight spaces and is a

name directive. It tells the **yum** module which packages it should ensure are up-to-date. The **yum** module's **name** directive (which is different from the task's name) can take a list of packages, which is indented ten spaces on the two following lines. After the list, the eight space indented **state** directive tells the **yum** module the latest version of the packages should be installed.

```
- name: latest version of httpd and firewalld installed
  yum:
    name:
      - httpd
      - firewalld
    state: latest
```

5. Define two more tasks in the play to ensure that **firewalld** is running and will start on boot, and will allow connections to the **http** service.
 - 5.1. Add the following lines to the **/home/student/imp-playbook/intranet.yml** file to create the task for ensuring that the **firewalld** service is enabled and running. Be sure to indent the line with four spaces, a dash, and a space. This indicates that the task is contained by the play and that it is an item in the **tasks** list.

The first entry provides a descriptive name for the task. The second entry is indented with eight spaces and calls the **service** module. The remaining entries are indented with ten spaces and pass the necessary arguments to ensure that the firewalld service is enabled and started.

```
- name: firewalld enabled and running
  service:
    name: firewalld
    enabled: true
    state: started
```

- 5.2. Add the following lines to the **/home/student/imp-playbook/intranet.yml** file to create the task to ensure **firewalld** opens the HTTP service to remote systems. Be sure to indent the line with four spaces, a dash, and a space. This indicates that the task is contained by the play and that it is an item in the **tasks** list.

The first entry provides a descriptive name for the task. The second entry is indented with six spaces and calls the **firewalld** module. The remaining entries are indented with eight spaces and pass the necessary arguments to ensure that access to the HTTP service is permanently allowed.

```
- name: firewalld permits http service
  firewalld:
    service: http
    permanent: true
    state: enabled
    immediate: yes
```

6. Add another task to the first play's list that ensures that the **httpd** service is running and will start at boot.

-
- 6.1. Add the following lines to the **/home/student/imp-playbook/intranet.yml** file to create the task to ensure the **httpd** service is enabled and running. Be sure to indent the line with four spaces, a dash, and a space. This indicates that the task is contained by the play and that it is an item in the **tasks** list.

The first entry provides a descriptive name for the task. The second entry is indented with six spaces and calls the **service** module. The remaining entries are indented with eight spaces and pass the necessary arguments to ensure that the httpd service is enabled and running.

```
- name: httpd enabled and running
  service:
    name: httpd
    enabled: true
    state: started
```

7. Add a final task to the first play's list that ensures that the correct content is in **/var/www/html/index.html**.
- 7.1. Add the following lines to the **/home/student/imp-playbook/intranet.yml** file to create the task that confirms the **/var/www/html/index.html** file is populated with the correct content. Be sure to indent the line with four spaces, a dash, and a space. This indicates that the task is contained by the play and that it is an item in the **tasks** list.

The first entry provides a descriptive name for the task. The second entry is indented with six spaces and calls the **copy** module. The remaining entries are indented with eight spaces and pass the necessary arguments to ensure that the right content is in the web page.

```
- name: test html page is installed
  copy:
    content: "Welcome to the example.com intranet!\n"
    dest: /var/www/html/index.html
```

8. In **/home/student/imp-playbook/intranet.yml**, define a second play targeted at **localhost** which will test the intranet web server. It does not need privilege escalation.

- 8.1. Add the following line to the **/home/student/imp-playbook/intranet.yml** file to denote the start of a second play.

```
- name: Test intranet web server
```

- 8.2. Add the following line to the **/home/student/imp-playbook/intranet.yml** file to indicate that the play applies to the **localhost** managed host. Be sure to indent the line with two spaces to indicate that it is contained by the second play.

```
  hosts: localhost
```

- 8.3. Add the following line to the `/home/student/imp-playbook/intranet.yml` file to disable privilege escalation. Be sure to align the indentation of the `become` directive with the `hosts` directive above it.

```
become: no
```

9. Add the following line to the `/home/student/imp-playbook/intranet.yml` file to define the beginning of the `tasks` list. Be sure to indent the line with two spaces to indicate that it is contained by the second play.

```
tasks:
```

10. Add a single task to the second play which uses the `uri` module to contact `http://servera.lab.example.com` and return successfully if the HTTP status code is 200.

- 10.1. Add the following lines to the `/home/student/imp-playbook/intranet.yml` file to create the task for verifying web services from the control node. Be sure to indent the first line with four spaces, a dash, and a space. This indicates that the task is an item in the second play's `tasks` list.

The first line provides a descriptive name for the task. The second line is indented with six spaces and calls the `uri` module. The remaining lines are indented with eight spaces and pass the necessary arguments to execute a query for web content from the control node to the managed host and verify the status code received.

```
- name: connect to intranet web server
  uri:
    url: http://servera.lab.example.com
    status_code: 200
```

11. Look at the final `/home/student/imp-playbook/intranet.yml` playbook and verify that it has the following structured content. Save the file.

```
---
- name: Enable intranet services
  hosts: servera.lab.example.com
  become: yes

  tasks:
    - name: latest version of httpd and firewalld installed
      yum:
        name:
          - httpd
          - firewalld
        state: latest

    - name: firewalld enabled and running
      service:
        name: firewalld
        enabled: true
        state: started

    - name: firewalld permits http service
      firewalld:
```

```
service: http
permanent: true
state: enabled
immediate: yes

- name: httpd enabled and running
  service:
    name: httpd
    enabled: true
    state: started

- name: test html page is installed
  copy:
    content: "Welcome to the example.com intranet!\n"
    dest: /var/www/html/index.html

- name: Test intranet web server
  hosts: localhost
  become: no

  tasks:
    - name: connect to intranet web server
      uri:
        url: http://servera.lab.example.com
        status_code: 200
```

12. Verify the syntax of the **intranet.yml** playbook by executing the **ansible-playbook** command with the **--syntax-check** option.

```
[student@workstation imp-playbook]$ ansible-playbook --syntax-check intranet.yml
playbook: intranet.yml
```

13. Execute the playbook. Read through the output generated to ensure that all tasks completed successfully.

```
[student@workstation imp-playbook]$ ansible-playbook intranet.yml
PLAY [Enable intranet services] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [latest version of httpd and firewalld installed] ****
changed: [servera.lab.example.com]

TASK [firewalld enabled and running] ****
ok: [servera.lab.example.com]

TASK [firewalld permits http service] ****
changed: [servera.lab.example.com]

TASK [httpd enabled and running] ****
changed: [servera.lab.example.com]

TASK [test html page is installed] ****
changed: [servera.lab.example.com]

PLAY [Test intranet web server] ****
```

```
TASK [Gathering Facts] ****
ok: [localhost]

TASK [connect to intranet web server] ****
ok: [localhost]

PLAY RECAP ****
localhost          : ok=2    changed=0    unreachable=0    failed=0
servera.lab.example.com  : ok=6    changed=4    unreachable=0    failed=0
```

14. Run **lab playbook grade** on **workstation** to grade your work.

```
[student@workstation imp-playbook]$ lab playbook grade
```

Lab: Implementing Playbooks

In this lab, you will configure and perform administrative tasks on managed hosts using a playbook.

Outcomes

You should be able to construct and execute a playbook to install, configure, and verify the status of web and database services on a managed host.

Before you begin

Log in as the **student** user on **workstation** and run **lab playbookinternet setup**. This setup script ensures that the managed host, **serverb.lab.example.com**, is reachable on the network. It also ensures that the correct Ansible configuration file and inventory are installed on the control node.

```
[student@workstation ~]$ lab playbookinternet setup
```

A developer responsible for the company's Internet website has asked you to write an Ansible playbook to automate the setup of his server environment on **serverb.lab.example.com**.

A working directory, **/home/student/imp-lab**, has been created on **workstation** for the Ansible project. The directory has already been populated with an **ansible.cfg** configuration file and an **inventory** inventory file. The managed host, **serverb.lab.example.com**, is already defined in this inventory file.

In this directory, create a playbook named **internet.yml**, which will contain two plays. The first play will require privilege escalation and must perform the following tasks in the specified order:

1. Use the **yum** module to ensure the latest versions of the following packages are installed: **firewalld**, **httpd**, **php**, **php-mysql**, and **mariadb-server**.
2. Ensure the **firewalld** service is enabled and started.
3. Ensure that **firewalld** is configured to allow connections to the ports used by the **httpd** service.
4. Ensure that the **httpd** service is enabled and started.
5. Ensure that the **mariadb** service is enabled and started.
6. Use the **get_url** module to ensure that the content at the URL **http://materials.example.com/grading/var/www/html/index.php** has been installed as the file **/var/www/html/index.php** on the managed host.

The second play does not require privilege escalation and should run a single task using the **uri** module to confirm that the URL **http://serverb.lab.example.com/** returns an HTTP status code of 200.

Following recommended practices, plays and tasks should have names that document their purpose, but this is not required. The example solution names plays and tasks.

Don't forget that you can use the **ansible-doc** command to get help with finding and using the modules for your tasks.

After the playbook is written, verify its syntax and then execute the playbook to implement the configuration. Verify your work by executing **lab playbookinternet grade**.



Note

The playbook used by this lab is very similar to the one you wrote in the preceding guided exercise in this chapter. If you don't want to create this lab's playbook from scratch, you can use that exercise's playbook as a starting point for this lab.

If you do, be careful to target the correct hosts and change the tasks to match the instructions for this exercise.

Steps

1. Change to the working directory, **/home/student/imp-lab**.
2. Create a new playbook, **/home/student/imp-lab/internet.yml**, and add the necessary entries to start a first play named "**Enable internet services**" and specify its intended managed host, **serverb.lab.example.com**. Also add an entry to enable privilege escalation.
3. Add the necessary entries to the **/home/student/imp-lab/internet.yml** file to define the tasks in the first play for configuring the managed host.
4. In **/home/student/imp-lab/internet.yml**, define another play for the task to be performed on the control node to test access to the web server that should be running on the **serverb** managed host. This play does not require privilege escalation.
5. Verify the syntax of the **internet.yml** playbook by using the **ansible-playbook** command.
6. Use **ansible-playbook** to run the playbook. Read through the output generated to ensure that all tasks completed successfully.
7. Run **lab playbookinternet grade** on **workstation** to grade your work.

```
[student@workstation imp-lab]$ lab playbookinternet grade
```

Solution

In this lab, you will configure and perform administrative tasks on managed hosts using a playbook.

Outcomes

You should be able to construct and execute a playbook to install, configure, and verify the status of web and database services on a managed host.

Before you begin

Log in as the **student** user on **workstation** and run **lab playbookinternet setup**. This setup script ensures that the managed host, **serverb.lab.example.com**, is reachable on the network. It also ensures that the correct Ansible configuration file and inventory are installed on the control node.

```
[student@workstation ~]$ lab playbookinternet setup
```

A developer responsible for the company's Internet website has asked you to write an Ansible playbook to automate the setup of his server environment on **serverb.lab.example.com**.

A working directory, **/home/student/imp-lab**, has been created on **workstation** for the Ansible project. The directory has already been populated with an **ansible.cfg** configuration file and an **inventory** inventory file. The managed host, **serverb.lab.example.com**, is already defined in this inventory file.

In this directory, create a playbook named **internet.yml**, which will contain two plays. The first play will require privilege escalation and must perform the following tasks in the specified order:

1. Use the **yum** module to ensure the latest versions of the following packages are installed: **firewalld**, **httpd**, **php**, **php-mysql**, and **mariadb-server**.
2. Ensure the **firewalld** service is enabled and started.
3. Ensure that **firewalld** is configured to allow connections to the ports used by the **httpd** service.
4. Ensure that the **httpd** service is enabled and started.
5. Ensure that the **mariadb** service is enabled and started.
6. Use the **get_url** module to ensure that the content at the URL **http://materials.example.com/grading/var/www/html/index.php** has been installed as the file **/var/www/html/index.php** on the managed host.

The second play does not require privilege escalation and should run a single task using the **uri** module to confirm that the URL **http://serverb.lab.example.com/** returns an HTTP status code of 200.

Following recommended practices, plays and tasks should have names that document their purpose, but this is not required. The example solution names plays and tasks.

Don't forget that you can use the **ansible-doc** command to get help with finding and using the modules for your tasks.

After the playbook is written, verify its syntax and then execute the playbook to implement the configuration. Verify your work by executing **lab playbookinternet grade**.



Note

The playbook used by this lab is very similar to the one you wrote in the preceding guided exercise in this chapter. If you don't want to create this lab's playbook from scratch, you can use that exercise's playbook as a starting point for this lab.

If you do, be careful to target the correct hosts and change the tasks to match the instructions for this exercise.

Steps

1. Change to the working directory, **/home/student/imp-lab**.

```
[student@workstation ~] cd /home/student/imp-lab
```

2. Create a new playbook, **/home/student/imp-lab/internet.yml**, and add the necessary entries to start a first play named "**Enable internet services**" and specify its intended managed host, **serverb.lab.example.com**. Also add an entry to enable privilege escalation.

- 2.1. Add the following entry to the beginning of **/home/student/imp-lab/internet.yml** to begin the YAML format.

```
---
```

- 2.2. Add the following entry to the **/home/student/imp-lab/internet.yml** file to denote the start of a play with a name of **Enable internet services**.

```
- name: Enable internet services
```

- 2.3. Add the following entry to the **/home/student/imp-lab/internet.yml** file to indicate that the play applies to the **serverb** managed host. Be sure to indent the entry with two spaces to indicate that it is contained by the play.

```
    hosts: serverb.lab.example.com
```

- 2.4. Add the following entry to the **/home/student/imp-lab/internet.yml** file to enable privilege escalation. Be sure to indent the entry with two spaces to indicate that it is contained by the play.

```
        become: yes
```

3. Add the necessary entries to the **/home/student/imp-lab/internet.yml** file to define the tasks in the first play for configuring the managed host.

- 3.1. Add the following entry to the **/home/student/imp-lab/internet.yml** file to define the beginning of the **tasks** list. Be sure to indent the entry with two spaces to indicate that it is contained by the play.

```
  tasks:
```

- 3.2. Add the following entry to the **/home/student/imp-lab/internet.yml** file to create a new task that ensures that the latest versions of the necessary packages are installed.

```
    - name: latest version of all required packages installed
      yum:
        name:
          - firewalld
          - httpd
          - mariadb-server
          - php
          - php-mysql
      state: latest
```

- 3.3. Add the necessary entries to the **/home/student/imp-lab/internet.yml** file to define the firewall configuration task.

```
    - name: firewalld enabled and running
      service:
        name: firewalld
        enabled: true
        state: started

    - name: firewalld permits http service
      firewalld:
        service: http
        permanent: true
        state: enabled
        immediate: yes
```

- 3.4. Add the necessary entries to the **/home/student/imp-lab/internet.yml** file to define the service management tasks.

```
    - name: httpd enabled and running
      service:
        name: httpd
        enabled: true
        state: started

    - name: mariadb enabled and running
      service:
        name: mariadb
        enabled: true
        state: started
```

- 3.5. Add the necessary entries to the **/home/student/imp-lab/internet.yml** file to define the final task for generating web content for testing.

```
- name: test php page is installed
  get_url:
    url: "http://materials.example.com/grading/var/www/html/index.php"
    dest: /var/www/html/index.php
    mode: 0644
```

4. In **/home/student/imp-lab/internet.yml**, define another play for the task to be performed on the control node to test access to the web server that should be running on the **serverb** managed host. This play does not require privilege escalation.

- 4.1. Add the following entry to the **/home/student/imp-lab/internet.yml** file to denote the start of a second play with a name of **Test internet web server**.

```
- name: Test internet web server
```

- 4.2. Add the following entry to the **/home/student/imp-lab/internet.yml** file to indicate that the play applies to the **localhost** managed host. Be sure to indent the entry with two spaces to indicate that it is contained by the play.

```
hosts: localhost
```

- 4.3. Add the following line after the **hosts** directive to disable privilege escalation for the second play. Match the indentation of the **hosts** line for the play.

```
become: no
```

- 4.4. Add an entry to the **/home/student/imp-lab/internet.yml** file to define the beginning of the **tasks** list. Be sure to indent the entry with two spaces to indicate that it is contained by the play.

```
tasks:
```

- 4.5. Add the following lines to the **/home/student/imp-lab/internet.yml** file to create the task for verifying the managed host's web services from the control node.

```
- name: connect to internet web server
  uri:
    url: http://serverb.lab.example.com
    status_code: 200
```

5. Verify the syntax of the **internet.yml** playbook by using the **ansible-playbook** command.

```
[student@workstation imp-lab]$ cat internet.yml
---
- name: Enable internet services
  hosts: serverb.lab.example.com
  become: yes
  tasks:
```

```

- name: latest version of all required packages installed
  yum:
    name:
      - firewalld
      - httpd
      - mariadb-server
      - php
      - php-mysql
    state: latest

- name: firewalld enabled and running
  service:
    name: firewalld
    enabled: true
    state: started

- name: firewalld permits http service
  firewalld:
    service: http
    permanent: true
    state: enabled
    immediate: yes

- name: httpd enabled and running
  service:
    name: httpd
    enabled: true
    state: started

- name: mariadb enabled and running
  service:
    name: mariadb
    enabled: true
    state: started

- name: test php page is installed
  get_url:
    url: "http://materials.example.com/grading/var/www/html/index.php"
    dest: /var/www/html/index.php
    mode: 0644

- name: Test internet web server
  hosts: localhost
  become: no

  tasks:
    - name: connect to internet web server
      uri:
        url: http://serverb.lab.example.com
        status_code: 200

[student@workstation imp-lab]$ ansible-playbook --syntax-check internet.yml
playbook: internet.yml

```

6. Use **ansible-playbook** to run the playbook. Read through the output generated to ensure that all tasks completed successfully.

```

[student@workstation imp-lab]$ ansible-playbook internet.yml
PLAY [Enable internet services] ****
TASK [Gathering Facts] ****

```

```
ok: [serverb.lab.example.com]

TASK [latest version of all required packages installed] ****
changed: [serverb.lab.example.com]

TASK [firewalld enabled and running] ****
ok: [serverb.lab.example.com]

TASK [firewalld permits http service] ****
changed: [serverb.lab.example.com]

TASK [httpd enabled and running] ****
changed: [serverb.lab.example.com]

TASK [mariadb enabled and running] ****
changed: [serverb.lab.example.com]

TASK [test php page installed] ****
changed: [serverb.lab.example.com]

PLAY [Test internet web server] ****

TASK [Gathering Facts] ****
ok: [localhost]

TASK [connect to internet web server] ****
ok: [localhost]

PLAY RECAP ****
localhost : ok=2    changed=0    unreachable=0    failed=0
serverb.lab.example.com : ok=7    changed=5    unreachable=0    failed=0
```

7. Run **lab playbookinternet grade** on **workstation** to grade your work.

```
[student@workstation imp-lab]$ lab playbookinternet grade
```

Summary

In this chapter, you learned:

- A *play* is an ordered list of tasks, which should be run against hosts selected from the inventory.
- A *playbook* is a text file that contains a list of one or more plays to run in order.
- Ansible playbooks are written in YAML format.
- YAML files are structured using space indentation to represent data hierarchy.
- Tasks are implemented using standardized code packaged as Ansible *modules*.
- The **ansible-doc** command can list installed modules, and provide documentation and example code snippets of how to use them in playbooks.
- The **ansible-playbook** command is used to verify playbook syntax and run playbooks.



CHAPTER 4

MANAGING VARIABLES AND INCLUSIONS

Overview	
Goal	To describe variable scope and precedence, manage variables and facts in a play, and manage inclusions.
Objectives	<ul style="list-style-type: none">• Manage variables in Ansible projects• Manage Facts in Playbooks• Include variables and tasks from external files into a playbook
Sections	<ul style="list-style-type: none">• Managing Variables (and Guided Exercise)• Managing Facts (and Guided Exercise)• Managing Inclusions (and Guided Exercise)
Lab	<ul style="list-style-type: none">• Lab: Managing Variables and Inclusions

Managing Variables

Objectives

After completing this section, students should be able to:

- Manage variables in Ansible projects

Introduction to Ansible Variables

Ansible supports variables that can be used to store values that can be reused throughout files in an entire Ansible project. This can help simplify creation and maintenance of a project and reduce the incidence of errors.

Variables provide a convenient way to manage dynamic values for a given environment in your Ansible project. Some examples of values that variables might contain include

- Users to create
- Packages to install
- Services to restart
- Files to remove
- Archives to retrieve from the Internet

Naming Variables

Variables have names which consist of a string that must start with a letter and can only contain letters, numbers, and underscores.

Consider the following table that shows the difference between invalid and valid variable names:

Ansible variable names	
Invalid variable names	Valid variable names
<code>web server</code>	<code>web_server</code>
<code>remote.file</code>	<code>remote_file</code>
<code>1st file</code>	<code>file_1</code> or <code>file1</code>
<code>remoteserver\$1</code>	<code>remote_server_1</code> or <code>remote_server1</code>

Defining Variables

Variables can be defined in a bewildering variety of places in an Ansible project. However, this can be simplified to three basic scope levels:

- *Global scope*: Variables set from the command line or Ansible configuration
- *Play scope*: Variables set in the play and related structures
- *Host scope*: Variables set on host groups and individual hosts by the inventory, fact gathering, or registered tasks

If the same variable name is defined at more than one level, the higher wins. So variables defined by the inventory are overridden by variables defined by the playbook, which are overridden by variables defined on the command line.

A detailed discussion of variable precedence is available in the Ansible documentation, a link to which is provided in the References at the end of this section.

Variables in Playbooks

Defining Variables in Playbooks

When writing playbooks, administrators can use their own variables and call them in a task. For example, a variable **web_package** can be defined with a value of **httpd** and called by the **yum** module in order to install the *httpd* package.

Playbook variables can be defined in multiple ways. One of the simplest is to place it in a **vars** block at the beginning of a playbook:

```
- hosts: all
  vars:
    user: joe
    home: /home/joe
```

It is also possible to define playbook variables in external files. In this case, instead of using **vars**, the **vars_files** directive may be used, followed by a list of external variable files relative to the playbook that should be read:

```
- hosts: all
  vars_files:
    - vars/users.yml
```

The playbook variables are then defined in that file or those files in YAML format:

```
user: joe
home: /home/joe
```

Using Variables in Playbooks

Once variables have been declared, administrators can use the variables in tasks. Variables are referenced by placing the variable name in double curly braces. Ansible substitutes the variable with its value when the task is executed.

```
vars:
  user: joe

tasks:
  # This line will read: Creates the user joe
  - name: Creates the user {{ user }}
    user:
      # This line will create the user named Joe
      name: "{{ user }}"
```



Important

When a variable is used as the first element to start a value, quotes are mandatory. This prevents Ansible from considering the variable as starting a YAML dictionary. The following message appears if quotes are missing:

```
yum:
  name: {{ service}}
    ^ here
We could be wrong, but this one looks like it might be an issue with
missing quotes. Always quote template expression brackets when they
start a value. For instance:

with_items:
  - {{ foo }}

Should be written as:

with_items:
  - "{{ foo }}"
```

Host Variables and Group Variables

Inventory variables that apply directly to hosts fall into two broad categories: *host variables* that apply to a specific host, and *group variables* that apply to all hosts in a host group or in a group of host groups. Host variables take precedence over group variables, but variables defined by a playbook take precedence over both.

One way to define host variables and group variables is to do it directly in the inventory file. This is an older approach and not preferred, but may be encountered by users:

- This is a host variable, **ansible_user**, being defined for the host **demo.example.com**.

```
[servers]
demo.example.com  ansible_user=joe
```

- In this example, a group variable **user** is being defined for the group **servers**.

```
[servers]
demo1.example.com
demo2.example.com

[servers:vars]
user=joe
```

- Finally, in this example a group variable **user** is being defined for the group **servers**, which happens to consist of two host groups each with two servers.

```
[servers1]
demo1.example.com
demo2.example.com

[servers2]
demo3.example.com
```

```
demo4.example.com

[servers:children]
servers1
servers2

[servers:vars]
user=joe
```

Among the disadvantages of this approach, it makes the inventory file more difficult to work with, mixes information about hosts and variables in the same file, and uses an obsolete syntax.

Using `group_vars` and `host_vars` Directories

The preferred approach is to create two directories in the same working directory as the inventory file or directory, **`group_vars`** and **`host_vars`**. These directories contain files defining group variables and host variables, respectively.



Important

The recommended practice is to define inventory variables using **`host_vars`** and **`group_vars`** directories, and not to define them directly in the inventory file or files.

To define group variables for the group **servers**, a YAML file named **`group_vars/servers`** would be created, and then the contents of that file would set variables to values using the same syntax as a playbook:

```
user: joe
```

Likewise, to define host variables for a particular host, a file with a name matching the host is created in **`host_vars`** to contain the host variables.

The following examples illustrate this approach in more detail. Consider the following scenario where there are two data centers to manage that has the following inventory file in **`~/project/inventory`**:

```
[admin@station project]$ cat ~/project/inventory
[datacenter1]
demo1.example.com
demo2.example.com

[datacenter2]
demo3.example.com
demo4.example.com

[datacenters:children]
datacenter1
datacenter2
```

- If a general value needs to be defined for all servers in both datacenters, a group variable can be set for **datacenters**:

```
[admin@station project]$ cat ~/project/group_vars/datacenters
package: httpd
```

- If the value to define varies for each datacenter, a group variable can be set for each datacenter:

```
[admin@station project]$ cat ~/project/group_vars/datacenter1
package: httpd
[admin@station project]$ cat ~/project/group_vars/datacenter2
package: apache
```

- If the value to be defined varies for each host in every datacenter, using host variables is recommended:

```
[admin@station project]$ cat ~/project/host_vars/demo1.example.com
package: httpd
[admin@station project]$ cat ~/project/host_vars/demo2.example.com
package: apache
[admin@station project]$ cat ~/project/host_vars/demo3.example.com
package: mariadb-server
[admin@station project]$ cat ~/project/host_vars/demo4.example.com
package: mysql-server
```

The directory structure for **project**, if it contained all of the example files above, might look like this:

```
project
├── ansible.cfg
├── group_vars
│   ├── datacenters
│   │   ├── datacenters1
│   │   └── datacenters2
│   └── host_vars
│       ├── demo1.example.com
│       ├── demo2.example.com
│       ├── demo3.example.com
│       └── demo4.example.com
└── inventory
    └── playbook.yml
```

Overriding Variables from the Command Line

Inventory variables are overridden by variables set in a playbook, but both kinds of variables may be overridden through arguments passed to the **ansible** or **ansible-playbook** commands on the command line. This can be useful in a case where the defined value for a variable needs to be overridden for a single host for a one-off run of a playbook. For example:

```
[user@demo ~]$ ansible-playbook main.yml --limit=demo2.example.com -e "package=apache"
```

Variables and Arrays

Instead of assigning a piece of configuration data that relates to the same element (a list of packages, a list of services, a list of users, etc.) to multiple variables, administrators can use *arrays*. One interesting consequence of this is that an array can be browsed.

For instance, consider the following snippet:

```
user1_first_name: Bob
```

```
user1_last_name: Jones
user1_home_dir: /users/bjones
user2_first_name: Anne
user2_last_name: Cook
user3_home_dir: /users/acook
```

This could be rewritten as an array called **users**:

```
users:
  bjones:
    first_name: Bob
    last_name: Jones
    home_dir: /users/bjones
  acook:
    first_name: Anne
    last_name: Cook
    home_dir: /users/acook
```

Users can then be accessed using the following variables:

```
# Returns 'Bob'
users.bjones.first_name

# Returns '/users/acook'
users.acook.home_dir
```

Because the variable is defined as a Python *dictionary*, an alternative syntax is available.

```
# Returns 'Bob'
users['bjones']['first_name']

# Returns '/users/acook'
users['acook']['home_dir']
```



Important

The dot notation can cause problems if the key names are the same as names of Python methods or attributes, such as **discard**, **copy**, **add**, and so on. Using the brackets notation can help avoid errors.

Both syntaxes are valid, but to make troubleshooting easier, it is recommended that one syntax used consistently in all files throughout any given Ansible project.

Registered Variables

Administrators can capture the output of a command by using the **register** statement. The output is saved into a variable that could be used later for either debugging purposes or in order to achieve something else, such as a particular configuration based on a command's output.

The following playbook demonstrates how to capture the output of a command for debugging purposes:

```
---
- name: Installs a package and prints the result
  hosts: all
```

```
tasks:
  - name: Install the package
    yum:
      name: httpd
      state: installed
      register: install_result

  - debug: var=install_result
```

When the playbook is run, the **debug** module is used to dump the value of the **install_result** registered variable to the terminal.

```
[user@demo ~]$ ansible-playbook playbook.yml
PLAY [Installs a package and prints the result] ****
TASK [setup] ****
ok: [demo.example.com]

TASK [Install the package] ****
ok: [demo.example.com]

TASK [debug] ****
ok: [demo.example.com] => {
    "install_result": {
        "changed": false,
        "msg": "",
        "rc": 0,
        "results": [
            "httpd-2.4.6-40.el7.x86_64 providing httpd is already installed"
        ]
    }
}

PLAY RECAP ****
demo.example.com    : ok=3      changed=0      unreachable=0      failed=0
```

Demonstration: Managing Variables

- From **workstation**, as the **student** user, change into the **~/demo_variables-playbook** directory.

```
[student@workstation ~]$ cd demo_variables-playbook
[student@workstation demo_variables-playbook]$
```

- Look at the **inventory** file. Notice that there are two host groups, **webservers** and **dbservers**, which are children of the larger host group **servers**. The **servers** host group has variables set the old way, directly in the inventory file, including one that sets **package** to **httpd**.

```
[webservers]
servera.lab.example.com

[dbservers]
servera.lab.example.com

[servers:children]
webservers
dbservers
```

```
[servers:vars]
ansible_user=devops
ansible_become=yes
package=httpd
```

3. Create a new playbook, **playbook.yml**, for all hosts. Using the **yum** module, install the package specified by the **package** variable.

```
---
- hosts: all
  tasks:
    - name: Installs the "{{ package }}" package
      yum:
        name: "{{ package }}"
        state: latest
```

4. Run the playbook using the **ansible-playbook** command. Watch the output as Ansible installs the *httpd* package.

```
[student@workstation demo_variables-playbook]$ ansible-playbook playbook.yml
PLAY ****
TASK [setup] ****
ok: [servera.lab.example.com]

TASK [Installs the "httpd" package] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com : ok=2    changed=1    unreachable=0    failed=0
```

5. Run an ad hoc command to ensure the *httpd* package has been successfully installed.

```
[student@workstation demo_variables-playbook]$ ansible servers \
> -a 'yum list installed httpd'
servera.lab.example.com | SUCCESS | rc=0 >>
Loaded plugins: langpacks, search-disabled-repos
Installed Packages
httpd.x86_64                  2.4.6-40.el7          @rhel_dvd
```

6. Create the **group_vars** directory and a new file **group_vars/dbservers** to set the variable **package** to **mariadb-server** for the **dbservers** host group in the recommended way.

```
[student@workstation demo_variables-playbook]$ cat group_vars/dbservers
package: mariadb-server
```

7. Run the playbook again using the **ansible-playbook** command. Watch Ansible install the *mariadb-server* package.

The **package** variable for the more specific host group **dbservers** took precedence over the one for its parent host group **servers**.

```
[student@workstation demo_variables-playbook]$ ansible-playbook playbook.yml
```

```
PLAY ****
TASK [setup] ****
ok: [servera.lab.example.com]

TASK [Installs the "mariadb-server" package] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com : ok=2    changed=1    unreachable=0    failed=0
```

The output indicates the variable defined for the hosts group has been overridden.

- Run an ad hoc command to confirm the **mariadb-server** package has been successfully installed.

```
[student@workstation demo_variables-playbook]$ ansible dbservers \
> -a 'yum list installed mariadb-server'
servera.lab.example.com | SUCCESS | rc=0 >>
Loaded plugins: langpacks, search-disabled-repos
Installed Packages
mariadb-server.x86_64          1:5.5.44-2.el7           @rhel_dvd
```

- For **servera.lab.example.com**, set the variable **package** to **screen**. Do this by creating a new directory, **host_vars**, and a file **host_vars/servera.lab.example.com** that sets the variable in the recommended way.

```
[student@workstation demo_variables-playbook]$ cat host_vars/servera.lab.example.com
package: screen
```

- Run the playbook again. Watch the output as Ansible installs the *screen* package.

The host-specific value of the **package** variable overrides any value set by the host's host groups.

```
[student@workstation demo_variables-playbook]$ ansible-playbook playbook.yml
PLAY ****
TASK [setup] ****
ok: [servera.lab.example.com]

TASK [Installs the "screen" package] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com : ok=2    changed=1    unreachable=0    failed=0
```

- Run an ad hoc command to confirm the **screen** package has been successfully installed.

```
[student@workstation demo_variables-playbook]$ ansible servera.lab.example.com \
> -a 'yum list installed screen'
servera.lab.example.com | SUCCESS | rc=0 >>
Loaded plugins: langpacks, search-disabled-repos
Installed Packages
screen.x86_64          4.1.0-0.21.20120314git3c2946.el7           rhel_dvd
```

12. Run the **ansible-playbook** command again, this time using the **-e** option to override the **package** variable.

```
[student@workstation demo_variables-playbook]$ ansible-playbook playbook.yml \
> -e 'package=mutt'
PLAY ****
TASK [setup] ****
ok: [servera.lab.example.com]

TASK [Installs the "mutt" package] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com : ok=2    changed=1    unreachable=0    failed=0
```

13. Run an ad hoc command to confirm the *mutt* package is installed.

```
[student@workstation demo_variables-playbook]$ ansible all \
> -a 'yum list installed mutt'
servera.lab.example.com | SUCCESS | rc=0 >>
Loaded plugins: langpacks, search-disabled-repos
Installed Packages
mutt.x86_64      5:1.5.21-26.el7          @rhel_dvd
```

References

- | Inventory – Ansible Documentation
http://docs.ansible.com/ansible/intro_inventory.html
- | Variables – Ansible Documentation
http://docs.ansible.com/ansible/playbooks_variables.htm
- | Variable Precedence: Where Should I Put A Variable?
http://docs.ansible.com/ansible/playbooks_variables.html#variable-precedence-where-should-i-put-a-variable
- | YAML Syntax – Ansible Documentation
<http://docs.ansible.com/ansible/YAMLSyntax.html>

Guided Exercise: Managing Variables

In this exercise, you will define and use variables in a playbook.

Outcomes

You should be able to:

- Define variables in a playbook.
- Create various tasks that include defined variables.

Before you begin

From **workstation**, run the lab setup script to confirm the environment is ready for the exercise to begin. The script creates the working directory, called **dev-vars-playbook**, and populates it with an Ansible configuration file and host inventory.

```
[student@workstation ~]$ lab manage-variables-playbooks setup
```

Steps

1. From **workstation**, as the **student** user, change into the **~/dev-vars-playbook** directory.

```
[student@workstation ~]$ cd ~/dev-vars-playbook  
[student@workstation dev-vars-playbook]$
```

2. Over the next several steps, you will create a playbook that installs the Apache web server and opens the ports for the service to be reachable. The playbook queries the web server to ensure it is up and running.

First, create the playbook **playbook.yml** and define the following variables in the **vars** section: **web_pkg**, which defines the name of the package to install for the web server; **firewall_pkg**, which defines the name of the firewall package; **web_service** for the name of the web service to manage; and **firewall_service** for the name of the firewall service to manage. Add the **python_pkg** variable to install a required package for the **uri** module; and **rule**, which defines the service to open.

```
- name: Deploy and start Apache HTTPD service  
hosts: webserver  
vars:  
  web_pkg: httpd  
  firewall_pkg: firewalld  
  web_service: httpd  
  firewall_service: firewalld  
  python_pkg: python-httplib2  
  rule: http
```

3. Create the **tasks** block and create the first task, which should use the **yum** module to make sure the latest versions of the required packages are installed.

```
tasks:  
  - name: Required packages are installed and up to date  
    yum:
```

```

name:
  - "{{ web_pkg }}"
  - "{{ firewall_pkg }}"
  - "{{ python_pkg }}"
state: latest

```



Note

If you use **ansible-doc yum** to look at the syntax for the **yum** module, you'll see that its **name** directive will take a list of packages that the module should work with, so you do not need separate tasks to make sure each package is up to date.

4. Create two tasks to make sure that the **httpd** and **firewalld** services are started and enabled.

```

- name: The {{ firewall_service }} service is started and enabled
  service:
    name: "{{ firewall_service }}"
    enabled: true
    state: started

- name: The {{ web_service }} service is started and enabled
  service:
    name: "{{ web_service }}"
    enabled: true
    state: started

```



Note

The **service** module works differently than the **yum** module, as documented by **ansible-doc service**. Its **name** directive takes the name of exactly one service to work with.

It is possible to write a single task that makes sure both of these services are started and enabled, by using the **with_items** directive that we'll cover later in this course.

5. Add a task that will ensure that certain content is in **/var/www/html/index.html**.

```

- name: Web content is in place
  copy:
    content: "Example web content"
    dest: /var/www/html/index.html

```

6. Add a task that will use the **firewalld** module to make sure the firewall ports are open for the **firewalld** service named in the **rule** variable.

```

- name: The firewall port for {{ rule }} is open
  firewalld:
    service: "{{ rule }}"
    permanent: true

```

```
immediate: true  
state: enabled
```

7. Create a new play that will query the web service to ensure everything has been correctly configured. It should run on **localhost**. Because of that fact, Ansible doesn't have to change identity, so set the **become** module to **false**. The **uri** module can be used to check a URL. For this task, check for a status code of 200 to confirm the server is running and properly configured.

```
- name: Verify the Apache service  
hosts: localhost  
become: false  
tasks:  
  - name: Ensure the webserver is reachable  
    uri:  
      url: http://servera.lab.example.com  
      status_code: 200
```

8. When completed, the playbook should appear as follows. Review the playbook and confirm that both plays are correct.

```
- name: Deploy and start Apache HTTPD service  
hosts: webserver  
vars:  
  web_pkg: httpd  
  firewall_pkg: firewalld  
  web_service: httpd  
  firewall_service: firewalld  
  python_pkg: python-httplib2  
  rule: http  
  
tasks:  
  - name: Required packages are installed and up to date  
    yum:  
      name:  
        - "{{ web_pkg }}"  
        - "{{ firewall_pkg }}"  
        - "{{ python_pkg }}"  
      state: latest  
  
  - name: The {{ firewall_service }} service is started and enabled  
    service:  
      name: "{{ firewall_service }}"  
      enabled: true  
      state: started  
  
  - name: The {{ web_service }} service is started and enabled  
    service:  
      name: "{{ web_service }}"  
      enabled: true  
      state: started  
  
  - name: Web content is in place  
    copy:  
      content: "Example web content"  
      dest: /var/www/html/index.html  
  
  - name: The firewall port for {{ rule }} is open  
    firewalld:  
      service: "{{ rule }}"
```

```

permanent: true
immediate: true
state: enabled

- name: Verify the Apache service
  hosts: localhost
  become: false
  tasks:
    - name: Ensure the webserver is reachable
      uri:
        url: http://servera.lab.example.com
        status_code: 200

```

- Before running the playbook, verify its syntax is correct by running **ansible-playbook --syntax-check**. If it reports any errors, correct them before moving to the next step. You should see output similar to the following:

```
[student@workstation dev-vars-playbook]$ ansible-playbook --syntax-check
playbook.yml

playbook: playbook.yml
```

- The playbook can be run using the **ansible-playbook** command. Watch the output as Ansible starts by installing the packages, starting and enabling the services, and ensuring the web server is reachable.

```
[student@workstation dev-vars-playbook]$ ansible-playbook playbook.yml

PLAY [Deploy and start Apache HTTPD service] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Required pacakges are installed and up to date] ****
changed: [servera.lab.example.com]

TASK [The firewalld service is started and enabled] ****
ok: [servera.lab.example.com]

TASK [The httpd service is started and enabled] ****
ok: [servera.lab.example.com]

TASK [Web content is in place] ****
changed: [servera.lab.example.com]

TASK [The firewall port for http is open] ****
ok: [servera.lab.example.com]

PLAY [Verify the Apache service] ****
TASK [Gathering Facts] ****
ok: [localhost]

TASK [Ensure the webserver is reachable] ****
ok: [localhost]

PLAY RECAP ****
localhost          : ok=2      changed=0      unreachable=0      failed=0
servera.lab.example.com : ok=6      changed=2      unreachable=0      failed=0
```

Cleanup

Run the **lab manage-variables-playbooks cleanup** command to undo the changes made to **servera**.

```
[student@workstation ~]$ lab manage-variables-playbooks cleanup
```

Managing Facts

Objectives

After completing this section, students should be able to:

- Manage Facts in Playbooks

Ansible Facts

Ansible *facts* are variables that are automatically discovered by Ansible on a managed host. Facts contain host-specific information that can be used just like regular variables in plays, conditionals, loops, or any other statement that depends on a value collected from a managed host.

Some of the facts gathered for a managed host might include

- The host name
- The kernel version
- The network interfaces
- The IP addresses
- The version of the operating system
- Various environment variables
- The number of CPUs
- The available or free memory
- The available disk space

Facts are a convenient way to retrieve the state of a managed host and to determine what action to take based on that state. For example:

- A server can be restarted by a conditional task which is run based on a fact containing the managed host's current kernel version.
- The MySQL configuration file can be customized depending on the available memory reported by a fact.
- The IPv4 address used in a configuration file can be set based on the value of a fact.

Normally, every play runs the **setup** module automatically before the first task in order to gather facts. This is reported as the **Gathering Facts** task in Ansible 2.3, or simply as **setup** in earlier versions of Ansible. You don't need to have a task to run **setup** in your play, it is automatically run for you.

In order to see what facts are gathered for managed hosts, you can run **setup** with an ad hoc command on those hosts. In the following example, an ad hoc command is used to run the **setup** module on the managed host **demo1.example.com**:

```
[user@demo ~]$ ansible demo1.example.com -m setup
ansible demo1.example.com -m setup
demo1.example.com | SUCCESS => {
    "ansible_facts": {
        "ansible_all_ipv4_addresses": [
            "172.25.250.10"
        ],
        "ansible_all_ipv6_addresses": [
            "fe80::5054:ff:fe00:fa0a"
        ],
        "ansible_architecture": "x86_64",
        "ansible_bios_date": "01/01/2011",
        "ansible_bios_version": "0.5.1",
        "ansible_cmdline": {
            "BOOT_IMAGE": "/boot/vmlinuz-3.10.0-327.el7.x86_64",
            "LANG": "en_US.UTF-8",
            "console": "ttyS0,115200n8",
            "crashkernel": "auto",
            "net.ifnames": "0",
            "no_timer_check": true,
            "ro": true,
            "root": "UUID=2460ab6e-e869-4011-acae-31b2e8c05a3b"
        }
    ...
    ... Output omitted ...
}
```

The output of the ad hoc command is returned in JSON format as a hash/dictionary of variables. You can browse the output to see what facts are gathered, and use them in your plays.

The following table shows some facts which might be gathered from a managed node that may be useful in a playbook:

Ansible Facts

Fact	Variable
Short hostname	<code>ansible_hostname</code>
Fully-qualified domain name	<code>ansible_fqdn</code>
Main IPv4 address (based on routing)	<code>ansible_default_ipv4.address</code>
A list of the names of all network interfaces	<code>ansible_interfaces</code>
Main disk first partition size (based on disk name, such as <code>vda</code> , <code>vdb</code> , and so on.)	<code>ansible_devices.vda.partitions.vda1.size</code>
A list of DNS servers	<code>ansible_dns.nameservers</code>
Version of the currently running kernel	<code>ansible_kernel</code>



Note

Remember that when a variable's value is a hash/dictionary, there are two syntaxes that can be used to retrieve the value. In the preceding example,

- `ansible_default_ipv4.address` can also be written
`ansible_default_ipv4['address']`
- `ansible_devices.vda.partitions.vda1.size` can also be written
`ansible_devices['vda']['partitions']['vda1']['size']`
- `ansible_dns.nameservers` can also be written
`ansible_dns['nameservers']`

When a fact is used in a playbook, Ansible dynamically substitutes the variable name with the corresponding value:

```
---
- hosts: all
  tasks:
    - name: Prints various Ansible facts
      debug:
        msg: >
          The default IPv4 address of {{ ansible_fqdn }}
          is {{ ansible_default_ipv4.address }}
```

The following output shows how Ansible was able to query the managed node and dynamically use the system information to update the variable. Moreover, facts can be also used to create dynamic groups of hosts that match particular criteria.

```
[user@demo ~]$ ansible-playbook playbook.yml
PLAY ****
TASK [Gathering Facts] ****
ok: [demo1.example.com]

TASK [Prints various Ansible facts] ****
ok: [demo1.example.com] => {
    "msg": "The default IPv4 address of demo1.example.com is
           172.25.250.10"
}

PLAY RECAP ****
demo1.example.com    : ok=2      changed=0      unreachable=0      failed=0
```

Turning Off Fact Gathering

Sometimes, you don't want to gather facts for your play. There are a couple of reasons why this might be the case. It might be that you are not using any facts and want to speed up the play or reduce load caused by the play on the managed hosts. It might be that the managed hosts can't run the `setup` module for some reason, or need to install some prerequisite software before gathering facts.

To disable fact gathering for a play, set the `gather_facts` key to `no`:

```
---
- name: This play gathers no facts automatically
  hosts: large_farm
  gather_facts: no
```

Even if **gather_facts: no** is set for a play, you can manually gather facts at any time by running a task that uses the **setup** module:

```
tasks:
- name: Manually gather facts
  setup:
```

Fact Filters

Ansible facts contain extensive information about the system. Administrators can use Ansible *filters* in order to limit the results returned when gathering facts from a managed node. Filters can be used to:

- Only retrieve information about network cards.
- Only retrieve information about disks.
- Only retrieve information about users.

To use filters, the expression needs to be passed as an option, using **-a 'filter=EXPRESSION'**. For example, to only return information about **eth0**, a filter can be applied on the **ansible_eth0** element:

```
[user@demo ~]$ ansible demo1.example.com -m setup -a 'filter=ansible_eth0'
demo1.lab.example.com | SUCCESS => {
  "ansible_facts": {
    "ansible_eth0": {
      "active": true,
      "device": "eth0",
      "ipv4": {
        "address": "172.25.250.10",
        "broadcast": "172.25.250.255",
        "netmask": "255.255.255.0",
        "network": "172.25.250.0"
      },
      "ipv6": [
        {
          "address": "fe80::5054:ff:fe00:fa0a",
          "prefix": "64",
          "scope": "link"
        }
      ],
      "macaddress": "52:54:00:00:fa:0a",
      "module": "virtio_net",
      "mtu": 1500,
      "pciid": "virtio0",
      "promisc": false,
      "type": "ether"
    }
  },
  "changed": false
}
```

Notice that Ansible only returns the facts in the **ansible_eth0** hash/dictionary.

Custom Facts

Administrators can create *custom facts* which are stored locally on each managed host. These facts are integrated into the list of standard facts gathered by **setup** when it runs on the managed host. These allow the managed host to provide arbitrary variables to Ansible which can be used to adjust the behavior of plays.

Custom facts can be defined in a static file, formatted as an INI file or using JSON. They can also be executable scripts which generate JSON output, just like a dynamic inventory script.

Custom facts allow an administrator to define certain values for managed hosts which plays might use to populate configuration files or conditionally run tasks. Dynamic custom facts allow the values for these facts or even which facts are provided to be determined programmatically when the play is run.

By default, **setup** loads custom facts from files and scripts in each managed host's **/etc/ansible/facts.d** directory. The name of each file or script must end in **.fact** in order to be used. Dynamic custom fact scripts must output JSON-formatted facts and must be executable.

This is an example of a static custom facts file written in INI format. An INI-formatted custom facts file contains a top level defined by a section, followed by the key-value pairs of the facts to define:

```
[packages]
web_package = httpd
db_package = mariadb-server

[users]
user1 = joe
user2 = jane
```

The same facts could be provided in JSON format. The following JSON facts are equivalent to the facts specified by the INI in the preceding example. The JSON data could be stored in a static text file or printed to standard output by an executable script:

```
{
  "packages": {
    "web_package": "httpd",
    "db_package": "mariadb-server"
  },
  "users": {
    "user1": "joe",
    "user2": "jane"
  }
}
```



Note

Custom fact files can not be in YAML format like a playbook. JSON format is the closest equivalent.

Custom facts are stored by **setup** in the **ansible_local** variable. Facts are organized based on the name of the file that defined them. For example, assume that the preceding custom facts are produced by a file saved as **/etc/ansible/facts.d/custom.fact** on the managed host. In that case, the value of **ansible_local['custom']['users']['user1']** is **joe**.

You can check the structure of your custom facts by running the **setup** module on the managed hosts with an ad hoc command.

```
[user@demo ~]$ ansible demo1.example.com -m setup -a 'filter=ansible_local'
demo1.lab.example.com | SUCCESS => {
    "ansible_facts": {
        "ansible_local": {
            "custom": {
                "packages": {
                    "db_package": "mariadb-server",
                    "web_package": "httpd"
                },
                "users": {
                    "user1": "joe",
                    "user2": "jane"
                }
            }
        },
        "changed": false
    }
}
```

Custom facts can be used the same way as default facts in playbooks:

```
[user@demo ~]$ cat playbook.yml
---
- hosts: all
  tasks:
    - name: Prints various Ansible facts
      debug:
        msg: >
          The package to install on {{ ansible_fqdn }}
          is {{ ansible_local.custom.packages.web_package }}

[user@demo ~]$ ansible-playbook playbook.yml
PLAY ****
TASK [Gathering Facts] ****
ok: [demo1.example.com]

TASK [Prints various Ansible facts] ****
ok: [demo1.example.com] => {
    "msg": "The package to install on demo1.example.com is httpd"
}

PLAY RECAP ****
demo1.example.com : ok=2     changed=0     unreachable=0    failed=0
```

Magic Variables

Some variables are not facts or configured through the **setup** module, but are also automatically set by Ansible. These *magic variables* can also be useful to get information specific to a particular managed host.

Four of the most useful are:

hostvars

Contains the variables for managed hosts, and can be used to get the values for another managed host's variables. It won't include the managed host's facts if they haven't been gathered yet for that host.

group_names

Lists all groups the current managed host is in.

groups

Lists all groups and hosts in the inventory.

inventory_hostname

Contains the hostname for the current managed host as configured in the inventory. This may be different from the hostname reported by facts for various reasons.

There are a number of other "magic variables" as well. For more information, see http://docs.ansible.com/ansible/playbooks_variables.html. One way to get insight into their values is to use the **debug** module to report on the contents of the **hostvars** variable for a particular host:

```
[user@demo ~]$ ansible localhost -m debug -a 'var=hostvars["localhost"]'
```



References

setup - Gathers facts about remote hosts – Ansible Documentation
http://docs.ansible.com/ansible/setup_module.html

Local Facts (Facts.d) – Variables – Ansible Documentation
http://docs.ansible.com/ansible/playbooks_variables.html#local-facts-facts-d

Guided Exercise: Managing Facts

In this exercise, you will gather Ansible facts from a managed host and use them in playbooks.

Outcomes

You should be able to:

- Gather facts from a host.
- Create various tasks that use the gathered facts.

Before you begin

From **workstation**, run the lab setup script to confirm the environment is ready for the exercise to begin. The script creates the working directory, **dev-vars-facts**, and populates it with an Ansible configuration file and host inventory.

```
[student@workstation ~]$ lab manage-variables-facts setup
```

Steps

1. From **workstation**, as the **student** user, change into the **~/dev-vars-facts** directory.

```
[student@workstation ~]$ cd ~/dev-vars-facts  
[student@workstation dev-vars-facts]$
```

2. The Ansible **setup** module retrieves facts from a system. Run an ad hoc command to retrieve the facts for all servers in the **webserver** group. The output will display all the facts gathered for **servera.lab.example.com** in JSON format. Review some of the variables displayed.

```
[student@workstation dev-vars-facts]$ ansible webserver -m setup  
... Output omitted ...  
servera.lab.example.com | SUCCESS => {  
    "ansible_facts": {  
        "ansible_all_ipv4_addresses": [  
            "172.25.250.10"  
        ],  
        "ansible_all_ipv6_addresses": [  
            "fe80::5054:ff:fe00:fa0a"  
        ],  
    }  
... Output omitted ...
```

3. Filter the facts matching the **ansible_user** expression. Append a wildcard to match all facts starting with **ansible_user**.

```
[student@workstation dev-vars-facts]$ ansible webserver -m setup -a  
'filter=ansible_user*'  
servera.lab.example.com | SUCCESS => {  
    "ansible_facts": {  
        "ansible_user_dir": "/root",  
        "ansible_user_gecos": "root",  
        "ansible_user_gid": 0,  
        "ansible_user_id": "root",  
        "ansible_user_shell": "/bin/bash",
```

```

        "ansible_user_uid": 0,
        "ansible_userspace_architecture": "x86_64",
        "ansible_userspace_bits": "64"
    },
    "changed": false
}

```

4. Create a fact file named **custom.fact**. The fact file defines the package to install and the service to start on **servera**. The file should read as follows:

```

[general]
package = httpd
service = httpd
state = started

```

5. Create the **setup_facts.yml** playbook to make the **/etc/ansible/facts.d** remote directory and to save the **custom.fact** file to that directory.

```

---
- name: Install remote facts
  hosts: webserver
  vars:
    remote_dir: /etc/ansible/facts.d
    facts_file: custom.fact
  tasks:
    - name: Create the remote directory
      file:
        state: directory
        recurse: yes
        path: "{{ remote_dir }}"
    - name: Install the new facts
      copy:
        src: "{{ facts_file }}"
        dest: "{{ remote_dir }}"

```

6. Run an ad hoc command with the **setup** module. Since user-defined facts are put into the **ansible_local** section, use a filter to display only this section. There should not be any custom facts at this point.

```

[student@workstation dev-vars-facts]$ ansible webserver -m setup -a
  'filter=ansible_local'
servera.lab.example.com | SUCCESS => {
    "ansible_facts": {},
    "changed": false
}

```

7. Before running the playbook, verify its syntax is correct by running **ansible-playbook --syntax-check**. If it reports any errors, correct them before moving to the next step. You should see output similar to the following:

```

[student@workstation dev-vars-facts]$ ansible-playbook --syntax-check
setup_facts.yml
playbook: setup_facts.yml

```

8. Run the **setup_facts.yml** playbook.

```
[student@workstation dev-vars-facts]$ ansible-playbook setup_facts.yml

PLAY [Install remote facts] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Create the remote directory] ****
changed: [servera.lab.example.com]

TASK [Install the new facts] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com      : ok=3      changed=2      unreachable=0      failed=0
```

9. To ensure the new facts have been properly installed, run an ad hoc command with the **setup** module again. Display only the **ansible_local** section. The custom facts should appear.

```
[student@workstation dev-vars-facts]$ ansible webserver -m setup -a
  'filter=ansible_local'
servera.lab.example.com | SUCCESS => {
    "ansible_facts": {
        "ansible_local": {
            "custom": {
                "general": {
                    "package": "httpd",
                    "service": "httpd",
                    "state": "started"
                }
            }
        },
        "changed": false
    }
}
```

10. It is now possible to create the main playbook that uses both default and user facts to configure **servera**. Over the next several steps, you will add to the playbook file. Start the **playbook.yml** playbook file with the following:

```
---
- name: Install Apache and starts the service
  hosts: webserver
```

11. Continue editing the **playbook.yml** file by creating the first task that installs the *httpd* package. Use the user fact for the name of the package.

```
tasks:
  - name: Install the required package
    yum:
      name: "{{ ansible_local.custom.general.package }}"
      state: latest
```

-
12. Create another task that uses the custom fact to start the **httpd** service.

```
- name: Start the service
  service:
    name: "{{ ansible_local.custom.general.service }}"
    state: "{{ ansible_local.custom.general.state }}"
```

13. When completed with all the tasks, the full playbook should look like the following. Review the playbook and ensure all the tasks are defined.

```
---
- name: Install Apache and starts the service
  hosts: webserver

  tasks:
    - name: Install the required package
      yum:
        name: "{{ ansible_local.custom.general.package }}"
        state: latest

    - name: Start the service
      service:
        name: "{{ ansible_local.custom.general.service }}"
        state: "{{ ansible_local.custom.general.state }}"
```

14. Before running the playbook, use an ad hoc command to verify the **httpd** service is not currently running on **servera**.

```
[student@workstation dev-vars-facts]$ ansible servera.lab.example.com -m command -a
'systemctl status httpd'
servera.lab.example.com | FAILED | rc=4 >>
Unit httpd.service could not be found.
```

15. Verify the syntax of the playbook by running **ansible-playbook --syntax-check**. If it reports any errors, correct them before moving to the next step. You should see output similar to the following:

```
[student@workstation dev-vars-facts]$ ansible-playbook --syntax-check playbook.yml
playbook: playbook.yml
```

16. Run the playbook using the **ansible-playbook** command. Watch the output as Ansible starts by installing the package, then enabling the service.

```
[student@workstation dev-vars-facts]$ ansible-playbook playbook.yml
PLAY [Install Apache and start the service] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Install the required package] ****
changed: [servera.lab.example.com]

TASK [Start the service] ****
```

```
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com : ok=3    changed=2    unreachable=0    failed=0
```

17. Use an ad hoc command to execute **systemctl** to check if the **httpd** service is now running on **servera**.

```
[student@workstation dev-vars-facts]$ ansible servera.lab.example.com -m command -a
'systemctl status httpd'
servera.lab.example.com | SUCCESS | rc=0 >>
● httpd.service - The Apache HTTP Server
  Loaded: loaded (/usr/lib/systemd/system/httpd.service; disabled; vendor preset:
disabled)
  Active: active (running) since Mon 2016-05-16 17:17:20 PDT; 12s ago
    Docs: man:httpd(8)
          man:apachectl(8)
  Main PID: 32658 (httpd)
    Status: "Total requests: 0; Current requests/sec: 0; Current traffic: 0 B/sec"
   CGroup: /system.slice/httpd.service
... Output omitted ...
```

Evaluation

From *workstation*, run the **lab manage-variables-facts grade** command to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab manage-variables-facts grade
```

Cleanup

Run the **lab manage-variables-facts** command to clean up the lab.

```
[student@workstation ~]$ lab manage-variables-facts cleanup
```

Managing Inclusions

After completing this section, students should be able to:

- Include variables and tasks from external files into a playbook

Inclusions

When working with complex or long playbooks, administrators can use separate files to divide tasks and lists of variables into smaller pieces for easier management. There are multiple ways to include task files and variables in a playbook.

- Tasks can be included in a playbook from an external file by using the **include** directive.

```
tasks:  
  - name: Include tasks to install the database server  
    include: tasks/db_server.yml
```

- The **include_vars** module can include variables defined in either JSON or YAML files, overriding host variables and playbook variables already defined.

```
tasks:  
  - name: Include the variables from a YAML or JSON file  
    include_vars: vars/variables.yml
```

Using multiple, external files for tasks and variables is a convenient way to build the main playbook in a modular way, and facilitates reuse of Ansible elements across multiple playbooks.

Including Tasks

Consider the following examples where it might be useful to manage a set of tasks as a file separate from the playbook:

- If fresh servers require complete configuration, administrators could create various sets of tasks for creating users, installing packages, configuring services, configuring privileges, setting up access to a shared file system, hardening the servers, installing security updates, and installing a monitoring agent. Each of these sets of tasks could be managed through a separate self-contained task file.
- If servers are managed collectively by the developers, the system administrators and the database administrators, every organization can write its own set of tasks that will be reviewed and integrated by the systems manager.
- If a server requires a particular configuration, it can be integrated as a set of tasks executed based on a conditional (that is, including the tasks only if specific criteria are met).
- If a group of servers need to run a particular task or set of tasks, the tasks might only be run on a server if it is part of a specific host group.

The **include** directive allows administrators to have a *task file* inserted at a particular point in a playbook. A task file is simply a file that contains a flat list of tasks:

```
- name: Installs the {{ package }} package
```

```
yum:
  name: "{{ package }}"
  state: latest

- name: Starts the {{ service }} service
  service:
    name: "{{ service }}"
    state: "{{ state }}"
```

The **include** directive is used in the playbook to specify what task file to include at what point in the playbook. A **vars** directive can be used to set variables used by the task file, and will override playbook variables, inventory variables, registered variables, and facts defined in the task file.

Assume the task file in the previous example was saved as the file **environment.yml**. An **include** directive with variables could be used in a playbook, like this:

```
---
- name: Install, start, and enable services
  hosts: all
  tasks:
    - name: Includes the tasks file and defines the variables
      include: environment.yml
      vars:
        package: mariadb-server
        service: mariadb
        state: started
      register: output

    - name: Debugs the included tasks
      debug:
        var: output
```

The following output shows how the tasks have been included and executed. In this case, the **mariadb-server** package has been installed because the variable **package** with a value of **mariadb-server** has been set by the **include** directive.

```
[user@demo ~]$ ansible-playbook.yml
PLAY [Install, start, and enable services] ****
TASK [setup] ****
ok: [demo1.example.com]

TASK [Includes the tasks file and defines the variables] ****
included: /home/student/demo-vars-inclusions/environment.yml for
          demo1.example.com

TASK [Installs the mariadb-server package] ****
ok: [demo1.example.com]

TASK [Starts the mariadb service] ****
changed: [demo1.example.com]

TASK [Debugs the included tasks] ****
ok: [demo1.example.com] => {
    "output": {
        "changed": false,
        "include": "environment.yml",
        "include_variables": {}
    }
}
```

```
}
```

```
PLAY RECAP ****
demo1.example.com      : ok=5      changed=1      unreachable=0      failed=0
```



Important

Tasks included in a playbook are evaluated based on their order in the playbook.

For example, a playbook may have a task to start a service provided by a software package, and a second task included from an external file that installs that software package. The external task must be included in the playbook before the task to start the package's service is declared in the playbook.

Administrators can create a dedicated directory for task files, and save all task files in that directory. Then the playbook simply includes task files from that directory in the playbook. This allows construction of a complex playbook while making it easier to manage its structure and components.



Note

For complex projects, *roles* provide a powerful way to organize included task files and playbooks. This topic will be discussed later in the course.

Including Variables

Just as tasks can be included in playbooks, variables can be externally defined and included in playbooks. There are many ways in which this can be done. Some of the ways to set variables include:

- Inventory variables defined in the inventory file or in external files in **host_vars** and **group_vars** directories
- Facts and registered variables
- Playbook variables defined in the playbook file with **vars** or in an external file through **vars_files**

The **include_vars** module is one more way to set variables in a playbook from an external file. The interesting thing about this method is that it is done through a module, and it overrides any values set through the above methods. This can be useful when combining task files that set values for their variables which you want to override, or in conjunction with conditional execution to set certain values only when specific conditions are met.

Included variable files may be defined using either JSON or YAML, YAML being the preferred syntax. The **include_vars** module takes the path of the variable file to import as an argument.



Important

If a task will require the variable settings from a variable file, the administrator must include the variable file in the playbook before the task is defined.

Consider the following snippets that demonstrate how to define variables and include them in a playbook with **include_vars**. The following YAML file, **variables.yml**, contains two variables that define the packages to install:

```
---
packages:
  web_package: httpd
  db_package: mariadb-server
```

To import these two variables in a playbook, the **include_vars** module can be used:

```
---
- name: Install web application packages
  hosts: all
  tasks:
    - name: Includes the tasks file and defines the variables
      include_vars: variables.yml

    - name: Debugs the variables imported
      debug:
        msg: >
          "{{ packages['web_package'] }} and {{ packages.db_package }}
          have been imported"
```

The following output shows that the variables have been successfully imported.

```
[user@prompt ~]$ ansible-playbook playbook.yml
PLAY [Install web application packages] ****
TASK [setup] ****
ok: [demo.example.com]

TASK [Includes the tasks file and defines the variables] ****
ok: [demo.example.com]

TASK [Debugs the variables imported] ****
ok: [demo.example.com] => {
    "msg": "httpd and mariadb-server have been imported"
}

PLAY RECAP ****
demo.example.com    : ok=3      changed=0      unreachable=0      failed=0
```



Important

When deciding where to define variables, try to keep things simple.

In most cases, it is not necessary to use **include_vars**, **host_vars** and **group_vars** files, playbook **vars_files** directives, inlined variables in playbook and inventory files, and command-line overrides all at the same time. In fact, that much complexity could make it hard to easily maintain your Ansible project.

- From **workstation**, as the **student** user, change into the **~/demo-vars-inclusions** directory.

```
[student@workstation ~]$ cd demo-vars-inclusions
[student@workstation demo-vars-inclusions]$
```

- Define the **paths.yml** variables file and create a dictionary that sets some system paths. Specify the **fileserver** base path as **/home/student/srv/filer/**, with the **ansible_fqdn** variable as the subdirectory. Specify the **dbpath** base path as **/home/student/srv/database/**, with the **ansible_fqdn** variable as the subdirectory.

```
---
paths:
  fileserver: /home/student/srv/filer/{{ ansible_fqdn }}
  dbpath: /home/student/srv/database/{{ ansible_fqdn }}
```

- Create the **fileservers.yml** playbook and include the **paths.yml** variables file. The **fileserver** will be created using the variable defined previously in the **paths.yml** variables file.

```
---
- name: Configure fileservers
  hosts: fileservers
  tasks:
    - name: Imports the variables file
      include_vars: paths.yml

    - name: Creates the remote directory
      file:
        path: "{{ paths.fileserver }}"
        state: directory
        mode: 0755
      register: result

    - name: Debugs the results
      debug:
        var: result
```

- Run the **fileservers.yml** playbook and examine the output.

```
[student@workstation demo-vars-inclusions]$ ansible-playbook fileservers.yml
PLAY [Configure fileservers] ****
TASK [setup] ****
```

```
ok: [servera.lab.example.com]

TASK [Imports the variables file] ****
ok: [servera.lab.example.com]

TASK [Creates the remote directory] ****
changed: [servera.lab.example.com]

TASK [Debugs the results] ****
ok: [servera.lab.example.com] => {
    "result": {
        "changed": true,
        "gid": 0,
        "group": "root",
        "mode": "0755",
        "owner": "root",
        "path": "/home/student/srv/filer/servera.lab.example.com",
        "secontext": "unconfined_u:object_r:user_home_t:s0",
        "size": 6,
        "state": "directory",
        "uid": 0
    }
}

PLAY RECAP ****
servera.lab.example.com      : ok=4      changed=1      unreachable=0      failed=0
```

The output shows the directory structure that has been created by Ansible, which matches the path that has been set by the **paths.fileservice** variable.

5. Create the **dbservers.yml** playbook. Include the variable file and use the **paths.dbpath** variable to create the directory structure.

```
---
- name: Configure DB Servers
  hosts: dbservers
  tasks:
    - name: Imports the variables file
      include_vars: paths.yml

    - name: Creates the remote directory
      file:
        path: "{{ paths.dbpath }}"
        state: directory
        mode: 0755
      register: result

    - name: Debugs the results
      debug:
        var: result
```

6. Run the **dbservers.yml** playbook and examine the output.

```
[student@workstation demo-vars-inclusions]$ ansible-playbook dbservers.yml
PLAY [Configure DB Servers] ****

TASK [setup] ****
ok: [servera.lab.example.com]

TASK [Imports the variables file] ****
```

```

ok: [servera.lab.example.com]

TASK [Creates the remote directory] ****
changed: [servera.lab.example.com]

TASK [Debugs the results] ****
ok: [servera.lab.example.com] => {
    "result": {
        "changed": true,
        "gid": 0,
        "group": "root",
        "mode": "0755",
        "owner": "root",
        "path": "/home/student/srv/database/servera.lab.example.com",
        "secontext": "unconfined_u:object_r:user_home_t:s0",
        "size": 6,
        "state": "directory",
        "uid": 0
    }
}

PLAY RECAP ****
servera.lab.example.com      : ok=4      changed=1      unreachable=0      failed=0

```

7. Create another variable file, **package.yml**, and define the name of the package to install.

```

---
packages:
  web_pkg: httpd

```

8. Create a task file, called **install_package.yml**, and create a basic task that installs a package.

```

---
- name: Installs {{ packages.web_pkg }}
  yum:
    name: "{{ packages.web_pkg }}"
    state: latest

```

9. Create the **playbook.yml** playbook. Define it for the hosts in the **fileservers** group. Include both variable files as well as the task file.

```

---
- name: Install fileserver packages
  hosts: fileservers
  tasks:
    - name: Includes the variable
      include_vars: package.yml

    - name: Installs the package
      include: install_package.yml

```

10. Run the playbook and watch the output.

```

[student@workstation demo-vars-inclusions]$ ansible-playbook playbook.yml
PLAY [Install fileserver packages] ****

```

```
TASK [setup] ****
ok: [servera.lab.example.com]

TASK [Includes the variable] ****
ok: [servera.lab.example.com]

TASK [Installs the package] ****
included: /home/student/demo-vars-inclusions/install_package.yml
for servera.lab.example.com

TASK [Installs httpd] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com : ok=4    changed=1    unreachable=0    failed=0
```

Note the *httpd* package has been installed from a task, whereas the name of the package to install is defined in the variables file.

11. Update the **playbook.yml** playbook to override the name of the package to install. Append a **vars** block to the **include** statement and define a dictionary to override the name of the package to install.

```
---
- name: Install fileserver packages
  hosts: fileservers
  tasks:
    - name: Includes the variable
      include_vars: package.yml

    - name: Installs the package
      include: install_package.yml
      vars:
        packages:
          web_pkg: tomcat
```

12. Run the playbook and watch the output as Ansible installs the *tomcat* package.

```
[student@workstation demo-vars-inclusions]$ ansible-playbook playbook.yml
PLAY [Install fileserver packages] ****

TASK [setup] ****
ok: [servera.lab.example.com]

TASK [Includes the variable] ****
ok: [servera.lab.example.com]

TASK [Installs the package] ****
included: /home/student/demo-vars-inclusions/install_package.yml
for servera.lab.example.com

TASK [Installs tomcat] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com : ok=4    changed=1    unreachable=0    failed=0
```



References

Playbook Roles and Include Statements – Ansible Documentation
http://docs.ansible.com/ansible/playbooks_roles.html

include_vars - Load variables from files, dynamically within a task – Ansible Documentation
http://docs.ansible.com/ansible/include_vars_module.html

file - Sets attributes of files – Ansible Documentation
http://docs.ansible.com/ansible/file_module.html

Guided Exercise: Managing Inclusions

In this exercise, you will manage inclusions in Ansible playbooks.

Outcomes

You should be able to:

- Define variables and tasks in separate files.
- Include variable files and task files in playbooks.

Before you begin

From **workstation**, run the lab setup script to confirm the environment is ready for the lab to begin. The script creates the working directory **dev-vars-inclusions**.

```
[student@workstation ~]$ lab manage-variables-inclusions setup
```

Steps

1. From **workstation**, as the **student** user, change into the directory **~/dev-vars-inclusions**.

```
[student@workstation ~]$ cd ~/dev-vars-inclusions  
[student@workstation dev-vars-inclusions]$
```

2. One task file, one variable file, and one playbook will be created for this exercise. The variable file defines, in YAML format, a variable used by the playbook. The task file defines the required tasks and includes variables that will be passed later on as arguments.
 - 2.1. Create a directory called **tasks** and change into that directory.

```
[student@workstation dev-vars-inclusions]$ mkdir tasks && cd tasks  
[student@workstation tasks]$
```

- 2.2. In the **tasks** directory, create the **environment.yml** task file. Define the two tasks that install and start the web server; use the **package** variable for the package name, **service** for the service name, and **svc_state** for the service state.

```
---  
  - name: Install the {{ package }} package  
    yum:  
      name: "{{ package }}"  
      state: latest  
  - name: Start the {{ service }} service  
    service:  
      name: "{{ service }}"  
      state: "{{ svc_state }}"
```

- 2.3. Change back into the main project directory. Create a directory named **vars** and change into that directory.

```
[student@workstation tasks]$ cd ..
```

```
[student@workstation dev-vars-inclusions]$ mkdir vars  
[student@workstation dev-vars-inclusions]$ cd vars  
[student@workstation vars]$
```

- 2.4. In the **vars** directory, create the **variables.yml** variables file. The file defines the **firewall_pkg** variable in YAML format. The file should read as follows:

```
---  
firewall_pkg: firewalld
```

- 2.5. Change back to the top-level project directory for the playbook.

```
[student@workstation vars]$ cd ..  
[student@workstation dev-vars-inclusions]$
```

3. Create and edit the main playbook, named **playbook.yml**. The playbook imports the tasks as well as the variables; and it installs the **firewalld** service and configures it.
- 3.1. Start with a name for the playbook, then add the **webserver** host group. Define a **rule** variable with a value of **http**.

```
---  
- name: Configure web server  
  hosts: webserver  
  vars:  
    rule: http
```

- 3.2. Continue editing the **playbook.yml** file. Define the first task, which uses the **include_vars** module to import extra variables in the playbook. The variables are used by other tasks in the playbook. Include the **variables.yml** variable file created previously.

```
tasks:  
  - name: Include the variables from the YAML file  
    include_vars: vars/variables.yml
```

- 3.3. Define the second task which uses the **include** module to include the base **environment.yml** playbook. Because the three defined variables are used in the base playbook, but are not defined, include a **vars** block. Set three variables in the **vars** section: **package** set as **httpd**, **service** set as **httpd**, and **svc_state** set as **started**.

```
- name: Include the environment file and set the variables  
  include: tasks/environment.yml  
  vars:  
    package: httpd  
    service: httpd  
    svc_state: started
```

- 3.4. Create three more tasks: one that installs the *firewalld* package, one that starts the **firewalld** service, and one that adds a rule for the HTTP service. Use the variables that were defined previously.

Create a task that installs the `firewalld` package using the `firewall_pkg` variable.

```
- name: Install the firewall
  yum:
    name: "{{ firewall_pkg }}"
    state: latest
```

Create a task that starts the `firewalld` service.

```
- name: Start the firewall
  service:
    name: firewalld
    state: started
    enabled: true
```

Create a task that adds a firewall rule for the HTTP service using the `rule` variable.

```
- name: Open the port for {{ rule }}
  firewalld:
    service: "{{ rule }}"
    immediate: true
    permanent: true
    state: enabled
```

3.5. Finally, add a task that creates the `index.html` file for the web server using the `copy` module. Create the file with the Ansible `ansible_fqdn` fact, which returns the fully qualified domain name. Also include a time stamp in the file using an Ansible fact. The task should read as follows:

```
- name: Create index.html
  copy:
    content: "{{ ansible_fqdn }} has been customized using Ansible on the
{{ ansible_date_time.date }}\n"
    dest: /var/www/html/index.html
```

3.6. When you have completed the main `playbook.yml` playbook, it should appear as follows:

```
---
- name: Configure web server
  hosts: webserver
  vars:
    rule: http
  tasks:
    - name: Include the variables from the YAML file
      include_vars: vars/variables.yml

    - name: Include the environment file and set the variables
      include: tasks/environment.yml
      vars:
        package: httpd
        service: httpd
        svc_state: started

    - name: Install the firewall
```

```
yum:
  name: "{{ firewall_pkg }}"
  state: latest

- name: Start the firewall
  service:
    name: firewalld
    state: started
    enabled: true

- name: Open the port for {{ rule }}
  firewalld:
    service: "{{ rule }}"
    immediate: true
    permanent: true
    state: enabled

- name: Create index.html
  copy:
    content: "{{ ansible_fqdn }} has been customized using Ansible on the
{{ ansible_date_time.date }}\n"
    dest: /var/www/html/index.html
```

4. Before running the playbook, verify its syntax is correct by running **ansible-playbook --syntax-check**. If it reports any errors, correct them before moving to the next step. You should see output similar to the following:

```
[student@workstation dev-vars-inclusions]$ ansible-playbook --syntax-check
playbook.yml

playbook: playbook.yml
```

5. Run the playbook using the **ansible-playbook** command. Watch the output as Ansible starts by including the **environment.yml** playbook and running its tasks, then keeps executing the tasks defined in the main playbook.

```
[student@workstation dev-vars-inclusions]$ ansible-playbook playbook.yml
PLAY [Configure web server] ****
ok: [servera.lab.example.com]

TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Include the variables from the YAML file] ****
ok: [servera.lab.example.com]

TASK [Install and start the web server] ****
changed: [servera.lab.example.com]

TASK [Start the service] ****
changed: [servera.lab.example.com]

TASK [Install the firewall] ****
changed: [servera.lab.example.com]

TASK [Start the firewall] ****
changed: [servera.lab.example.com]

TASK [Open the port for http] ****
changed: [servera.lab.example.com]
```

```
TASK [Create index.html] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com    : ok=9      changed=4      unreachable=0      failed=0
```

6. Use **curl** to ensure the web server is reachable from **workstation**. Because the **index.html** has been created, the output should appear as follows:

```
[student@workstation dev-vars-inclusions]$ curl http://servera.lab.example.com
servera.lab.example.com has been customized using Ansible on the 2017-07-21
```

Evaluation

Run the **lab manage-variables-inclusions** command from **workstation** to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab manage-variables-inclusions grade
```

Cleanup

Run the **lab manage-variables-inclusions cleanup** command to clean up after the lab.

```
[student@workstation ~]$ lab manage-variables-inclusions cleanup
```

Lab: Managing Variables and Inclusions

In this lab, you will deploy a database as well as a web server. For this lab, various Ansible modules, variables, and tasks will be defined.

Outcomes

You should be able to:

- Define variables in a playbook.
- Create custom facts for a managed host and use these facts in the main playbook.
- Create a separate set of tasks to include in the main playbook.
- Define a variable in a variable file and include the variable file in the main playbook.

Before you begin

From **workstation.lab.example.com**, open a new terminal and run the setup script to prepare the environment. The script creates the project directory, called **lab-managing-vars**, and populates it with an Ansible configuration file and host inventory.

```
[student@workstation ~]$ lab manage-variables setup
```

Steps

1. Create all of the files related to this lab in, or below, the **lab-managing-vars** project directory.

2. Defining custom facts

Create a facts file in INI format called **custom.fact**. Create a section called **packages** and define two facts: one called **db_package**, with a value of **mariadb-server**, and one called **web_package**, with a value of **httpd**. Create a section called **services** with two facts: one called **db_service**, with a value of **mariadb**, and one called **web_service**, with a value of **httpd**.

Define a playbook, called **setup_facts.yml**, that will install the facts on **serverb**.

3. Installing facts

Run the playbook to install the custom facts and verify the facts are available as Ansible facts.

4. Defining variables

Create a directory for variables, called **vars**. Define a YAML variable file, called **main.yml**, in that directory that defines a new variable, called **web_root**, with a value of **/var/www/html**.

5. Defining tasks

From the project directory, **lab-managing-vars**, create a directory for tasks, called **tasks**. Define a task file in the subdirectory, called **main.yml**, that instructs Ansible to

install both the web server package and the database package using facts Ansible gathered from **serverb.lab.example.com**. When they are installed, start the two services.

6. Defining the main playbook

Create the main playbook, **playbook.yml**, in the top-level directory for this lab. The playbook should be in the following order: target the **lamp** hosts group and define a new variable, **firewall**, with a value of **firewalld**.

Create the following tasks:

- A task that includes the variable file, **main.yml**.
- A task that includes the tasks defined in the tasks file.
- A task for installing the latest version of the *firewall* package.
- A task for starting the **firewall** service.
- A task for opening TCP port 80 permanently.
- A task that uses the **copy** module to create the **index.html** page in the directory the variable defines.

The **index.html** should appear as follows:

```
serverb.lab.example.com (172.25.250.11) has been customized by Ansible
```

Both the host name and the IP address should use Ansible facts.

7. Running the playbook

Run the playbook **playbook.yml** created in the previous step.

8. Testing the deployment

From **workstation**, use **curl** to ensure the web server is reachable. Also use an ad hoc command to connect to **serverb** as the **devops** user and ensure the **mariadb** service is running.

Evaluation

Run the **lab manage-variables grade** command from *workstation* to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab manage-variables grade
```

Cleanup

Run the **lab manage-variables cleanup** command to cleanup after the lab.

```
[student@workstation ~]$ lab manage-variables cleanup
```

Solution

In this lab, you will deploy a database as well as a web server. For this lab, various Ansible modules, variables, and tasks will be defined.

Outcomes

You should be able to:

- Define variables in a playbook.
- Create custom facts for a managed host and use these facts in the main playbook.
- Create a separate set of tasks to include in the main playbook.
- Define a variable in a variable file and include the variable file in the main playbook.

Before you begin

From **workstation.lab.example.com**, open a new terminal and run the setup script to prepare the environment. The script creates the project directory, called **lab-managing-vars**, and populates it with an Ansible configuration file and host inventory.

```
[student@workstation ~]$ lab manage-variables setup
```

Steps

1. Create all of the files related to this lab in, or below, the **lab-managing-vars** project directory.

```
[student@workstation ~]$ cd ~/lab-managing-vars  
[student@workstation lab-managing-vars]$
```

2. **Defining custom facts**

Create a facts file in INI format called **custom.fact**. Create a section called **packages** and define two facts: one called **db_package**, with a value of **mariadb-server**, and one called **web_package**, with a value of **httpd**. Create a section called **services** with two facts: one called **db_service**, with a value of **mariadb**, and one called **web_service**, with a value of **httpd**.

Define a playbook, called **setup_facts.yml**, that will install the facts on **serverb**.

- 2.1. Create the fact file **custom.fact**. The file should appear as follows:

```
[packages]  
db_package = mariadb-server  
web_package = httpd  
  
[services]  
db_service = mariadb  
web_service = httpd
```

- 2.2. From the project directory, create the **setup_facts.yml** playbook to install the facts on the managed host, **serverb.lab.example.com**. Use the **file** and **copy** modules to install the custom facts. The playbook should appear as follows:

```
---
- name: Install remote facts
  hosts: lamp
  vars:
    remote_dir: /etc/ansible/facts.d
    facts_file: custom.fact
  tasks:
    - name: Create the remote directory
      file:
        state: directory
        recurse: yes
        path: "{{ remote_dir }}"
    - name: Install the new facts
      copy:
        src: "{{ facts_file }}"
        dest: "{{ remote_dir }}"
```

3. Installing facts

Run the playbook to install the custom facts and verify the facts are available as Ansible facts.

- Before running the playbook, verify its syntax is correct by running **ansible-playbook --syntax-check**. If it reports any errors, correct them before moving to the next step. You should see output similar to the following:

```
[student@workstation lab-managing-vars]$ ansible-playbook --syntax-check
setup_facts.yml

playbook: setup_facts.yml
```

- Run the playbook.

```
[student@workstation lab-managing-vars]$ ansible-playbook setup_facts.yml
PLAY [Install remote facts] ****
TASK [Gathering Facts] ****
ok: [serverb.lab.example.com]

TASK [Create the remote directory] ****
changed: [serverb.lab.example.com]

TASK [Install the new facts] ****
changed: [serverb.lab.example.com]

PLAY RECAP ****
serverb.lab.example.com    : ok=3      changed=2      unreachable=0      failed=0
```

- Ensure the newly created facts can be retrieved.

```
[student@workstation lab-managing-vars]$ ansible lamp -m setup -a
'filter=ansible_local'
serverb.lab.example.com | SUCCESS => {
  "ansible_facts": {
    "ansible_local": {
      "custom": {
```

```

    "packages": {
        "db_package": "mariadb-server",
        "web_package": "httpd"
    },
    "services": {
        "db_service": "mariadb",
        "web_service": "httpd"
    }
},
"changed": false
}

```

4. Defining variables

Create a directory for variables, called **vars**. Define a YAML variable file, called **main.yml**, in that directory that defines a new variable, called **web_root**, with a value of **/var/www/html**.

- 4.1. Create the variables directory, **vars**, inside the project directory.

```
[student@workstation lab-managing-vars]$ mkdir vars
```

- 4.2. Create the variables file **vars/main.yml**. The file should contain the following content:

```

---
web_root: /var/www/html

```

5. Defining tasks

From the project directory, **lab-managing-vars**, create a directory for tasks, called **tasks**. Define a task file in the subdirectory, called **main.yml**, that instructs Ansible to install both the web server package and the database package using facts Ansible gathered from **serverb.lab.example.com**. When they are installed, start the two services.

- 5.1. Create the **tasks** directory inside the project directory.

```
[student@workstation lab-managing-vars]$ mkdir tasks
```

- 5.2. Create the tasks file, **tasks/main.yml**. The tasks should install both the database and the web server, and start the services **httpd** and **mariadb**. Use the custom Ansible facts for the name of the services to manage. The file should appear as follows:

```

---
- name: Install and start the database and web servers
  yum:
    name:
      - "{{ ansible_local.custom.packages.db_package }}"
      - "{{ ansible_local.custom.packages.web_package }}"
    state: latest

  - name: Start the database service
    service:
      name: "{{ ansible_local.custom.services.db_service }}"

```

```
state: started
enabled: true

- name: Start the web service
  service:
    name: "{{ ansible_local.custom.services.web_service }}"
    state: started
    enabled: true
```

6. Defining the main playbook

Create the main playbook, **playbook.yml**, in the top-level directory for this lab. The playbook should be in the following order: target the **lamp** hosts group and define a new variable, **firewall**, with a value of **firewalld**.

Create the following tasks:

- A task that includes the variable file, **main.yml**.
- A task that includes the tasks defined in the tasks file.
- A task for installing the latest version of the *firewall* package.
- A task for starting the **firewall** service.
- A task for opening TCP port 80 permanently.
- A task that uses the **copy** module to create the **index.html** page in the directory the variable defines.

The **index.html** should appear as follows:

```
serverb.lab.example.com (172.25.250.11) has been customized by Ansible
```

Both the host name and the IP address should use Ansible facts.

6.1. The following steps edit the single **playbook.yml** file.

Create the main playbook, **playbook.yml**, for the hosts in the **lamp** hosts group, and define the **firewall** variable.

```
---
- name: Install and configure lamp
  hosts: lamp
  vars:
    firewall: firewalld
```

6.2. Add the **tasks** block and define the first task that includes the variables file located under **vars/main.yml**.

```
tasks:
  - name: Include the variable file
    include_vars: vars/main.yml
```

6.3. Create the second task, which imports the tasks file located under **tasks/main.yml**.

```
- name: Include the tasks
  include: tasks/main.yml
```

- 6.4. Create the tasks that install the firewall, start the service, open port 80, and reload the service.

```
- name: Install the firewall
  yum:
    name: "{{ firewall }}"
    state: latest

- name: Start the firewall
  service:
    name: "{{ firewall }}"
    state: started
    enabled: true

- name: Open the port for the web server
  firewalld:
    service: http
    state: enabled
    immediate: true
    permanent: true
```

- 6.5. Finally, create the task that uses the **copy** module to create a custom main page, **index.html**. Use the variable **web_root**, defined in the variables file, for the home directory of the web server.

```
- name: Create index.html
  copy:
    content: "{{ ansible_fqdn }}({{ ansible_default_ipv4.address }}) has
    been customized by Ansible\n"
    dest: "{{ web_root }}/index.html"
```

- 6.6. When complete, the tree should appear as follows:

```
[student@workstation lab-managing-vars]$ tree
.
├── ansible.cfg
├── custom.fact
├── inventory
├── playbook.yml
├── setup_facts.yml
└── tasks
    └── main.yml
└── vars
    └── main.yml

2 directories, 7 files
```

- 6.7. The main playbook should appear as follows:

```
---
- name: Install and configure lamp
  hosts: lamp
```

```
vars:
  firewall: firewalld

tasks:
  - name: Include the variable file
    include_vars: vars/main.yml

  - name: Include the tasks
    include: tasks/main.yml

  - name: Install the firewall
    yum:
      name: "{{ firewall }}"
      state: latest

  - name: Start the firewall
    service:
      name: "{{ firewall }}"
      state: started
      enabled: true

  - name: Open the port for the web server
    firewalld:
      service: http
      state: enabled
      immediate: true
      permanent: true

  - name: Create index.html
    copy:
      content: "{{ ansible_fqdn }}({{ ansible_default_ipv4.address }}) has
      been customized by Ansible\n"
      dest: "{{ web_root }}/index.html"
```

7. Running the playbook

Run the playbook **playbook.yml** created in the previous step.

- Before running the playbook, verify its syntax is correct by running **ansible-playbook --syntax-check**. If it reports any errors, correct them before moving to the next step. You should see output similar to the following:

```
[student@workstation lab-managing-vars]$ ansible-playbook --syntax-check
playbook.yml
```

- Using the **ansible-playbook** command, run the playbook.

```
[student@workstation lab-managing-vars]$ ansible-playbook playbook.yml
PLAY [Install and configure lamp]*****
...
... Output omitted ...

PLAY RECAP *****
serverb.lab.example.com      : ok=9      changed=5      unreachable=0      failed=0
```

8. Testing the deployment

From **workstation**, use **curl** to ensure the web server is reachable. Also use an ad hoc command to connect to **serverb** as the **devops** user and ensure the **mariadb** service is running.

- 8.1. From **workstation**, use **curl** to ensure the web server has been successfully started and it is reachable. If the following message appears, it indicates that the web server has been installed, the firewall has been updated with a new rule and the included variable has been successfully used.

```
[student@workstation lab-managing-vars]$ curl http://serverb  
serverb.lab.example.com(172.25.250.11) has been customized by Ansible
```

- 8.2. Use an ad hoc Ansible command to ensure the **mariadb** service is running on **serverb.lab.example.com**.

```
[student@workstation lab-managing-vars]$ ansible lamp -a 'systemctl status  
mariadb'  
serverb.lab.example.com | SUCCESS | rc=0 >>  
● mariadb.service - MariaDB database server  
    Loaded: loaded (/usr/lib/systemd/system/mariadb.service; disabled; vendor  
    preset: disabled)  
    Active: active (running) since Fri 2016-04-01 10:50:40 PDT; 7min ago  
... Output omitted ...
```

Evaluation

Run the **lab manage-variables grade** command from *workstation* to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab manage-variables grade
```

Cleanup

Run the **lab manage-variables cleanup** command to cleanup after the lab.

```
[student@workstation ~]$ lab manage-variables cleanup
```

Summary

In this chapter, you learned:

- Ansible *variables* allow administrators to reuse values across files in an entire Ansible project
- Variables have names which consist of a string that must start with a letter and can only contain letters, numbers, and underscores
- Variables can be defined for hosts and host groups in the inventory, for playbooks, by facts and external files, and from the command line
- It is better to store inventory variables in files in the **host_vars** and **group_vars** directory relative to the inventory than in the inventory file itself
- Ansible *facts* are variables that are automatically discovered by Ansible from a managed host
- In a playbook, when a variable is used at the start of a value, quotes are mandatory
- The **register** keyword can be used to capture the output of a command in a variable.
- Both **include** and **include_vars** modules can be used to include tasks or variable files in YAML or JSON format in playbooks.



CHAPTER 5

IMPLEMENTING TASK CONTROL

Overview	
Goal	Manage task control, handlers, and tags in Ansible playbooks
Objectives	<ul style="list-style-type: none">• Construct conditionals and loops in a playbook• Implement handlers in a playbook• Implement tags in a playbook• Resolve errors in a playbook
Sections	<ul style="list-style-type: none">• Constructing Flow Control (and Guided Exercise)• Implementing Handlers (and Guided Exercise)• Implementing Tags (and Guided Exercise)• Handling Errors (and Guided Exercise)
Lab	<ul style="list-style-type: none">• Implementing Task Control

Constructing Flow Control

Objectives

After completing this section, students should be able to:

- Implement loops in playbooks
- Construct conditionals in playbooks
- Combine loops and conditionals

Task Iteration with Loops

Ansible supports several different ways to iterate a task over a set of items using a loop. Loops can repeat a task using each item in a list, the contents of each of the files in a list, a generated sequence of numbers, or using more complicated structures.

Using loops saves administrators from the need to write multiple tasks that use the same module. For example, instead of writing five tasks to ensure five users exist, one task can be written that iterates over a list of five users to ensure they all exist.

Simple Loops

A simple loop iterates a task over a list of items. The **with_items** key is added to the task, and takes as a value the list of items over which the task should be iterated. The loop variable **item** holds the current value being used for this iteration.

Consider the following snippet that uses the **service** module twice in order to ensure two network services are running:

```
- name: Postfix is running
  service:
    name: postfix
    state: started

- name: Dovecot is running
  service:
    name: dovecot
    state: started
```

These two tasks can be rewritten to use a simple loop, so that only one task is needed to ensure both services are running:

```
- name: Postfix and Dovecot are running
  service:
    name: "{{ item }}"
    state: started
  with_items:
    - postfix
    - dovecot
```

The list used by **with_items** can be provided by a variable. In the following example, the variable **mail_services** contains the list of services that need to be running.

```
vars:
```

```

mail_services:
  - postfix
  - dovecot

tasks:
  - name: Postfix and Dovecot are running
    service:
      name: "{{ item }}"
      state: started
    with_items: "{{ mail_services }}"

```

Simple Loops over Lists of Hash/Dictionaries

The **with_items** list does not need to be a list of simple values. In the following example, each item in the list is actually a hash/dictionary. Each hash/dictionary in the example has two keys, **name** and **groups**, and the value of each key in the current **item** loop variable can be retrieved with the **item.name** and **item.groups** variables, respectively.

```

- name: Users exist and are in the correct groups
  user:
    name: "{{ item.name }}"
    state: present
    groups: "{{ item.groups }}"
  with_items:
    - { name: 'jane', groups: 'wheel' }
    - { name: 'joe', groups: 'root' }

```

The outcome of the preceding task is that user **jane** is present and a member of group **wheel**, and that user **joe** is present and a member of group **root**.

Nested Loops

The **with_nested** key is used for nested loops, loops run inside of loops. It takes a list of two or more lists. For example, given a list of two lists, the task will iterate each item in the first list in combination with each item in the second list.

To illustrate this better, look at the following snippet from a play. The **mysql_user** module is used to grant all users in the first list all MySQL privileges to all tables of all databases named in the second list, using nested loops.

```

tasks:
  - name: All DB users have privileges on all databases
    mysql_user:
      name: "{{ item[0] }}"
      priv: "{{ item[1] }}.*:ALL"
      append_privs: yes
      password: redhat
    with_nested:
      - [ 'joe', 'jane' ]
      - [ 'clientdb', 'employeedb', 'providerdb' ]

```

The preceding example iterates six times. It adds privileges for user **joe** to each of the three databases, one at a time, and then does the same for user **jane**.

The list items in the **with_nested** list can be defined in variables. The previous example can be rewritten to use variables to contain the lists of database users and databases:

```

vars:
  db_users:
    - joe

```

```

- jane

databases:
  - clientdb
  - employeedb
  - providerdb

tasks:
  - name: All DB users have privileges on all databases
    mysql_user:
      name: "{{ item[0] }}"
      priv: "{{ item[1] }}.*:ALL"
      append_privs: yes
      password: redhat
    with_nested:
      - "{{ db_users }}"
      - "{{ databases }}"

```

Other Common Loop Directives

The following table shows some additional types of loops supported by Ansible.

Ansible Loops

Loop Keyword	Description
with_file	Takes a list of control node file names. item is set to the content of each file in sequence.
with_fileglob	Takes a file name globbing pattern. item is set to each file in a directory on the control node that matches that pattern, in sequence, non-recursively.
with_sequence	Generates a sequence of items in increasing numerical order. Can take start and end arguments which have a decimal, octal, or hexadecimal integer value.
with_random_choice	Takes a list. item is set to one of the list items at random.

Running Tasks Conditionally

Ansible can use *conditionals* to execute tasks or plays when certain conditions are met. For example, a conditional can be used to determine the available memory on a managed host before Ansible installs or configures a service.

Conditionals allow administrators to differentiate between managed hosts and assign them functional roles based on the conditions that they meet. Playbook variables, registered variables, and Ansible facts can all be tested with conditionals. Operators such as string comparison, mathematical operators, and Booleans are available.

The following examples illustrate some ways in which conditionals can be used by Ansible.

- A hard limit can be defined in a variable (for example, **min_memory**) and compared against the available memory on a managed host.
- The output of a command can be captured and evaluated by Ansible to determine whether or not a task completed before taking further action. For example, if a program fails, then a batch will need to be skipped.
- Ansible facts can be used to determine the managed host network configuration and decide which template file to send (for example, network bonding or trunking).

- The number of CPUs can be evaluated to determine how to properly tune a web server.
- Registered variables can be compared with defined variables to check a service change. For example, this can be used to verify the MD5 checksum of a file.

Ansible **when** Statement

The **when** statement is used to run a task conditionally. It takes as a value the condition to test. If the condition is met, the task runs. If the condition is not met, the task is skipped.

One of the simplest conditions which can be tested is whether a Boolean variable is true or false. The **when** statement in the following example causes the task to run only if **run_my_task** is true:

```
---
- hosts: all
  vars:
    run_my_task: true

  tasks:
    - name: httpd package is installed
      yum:
        name: httpd
      when: run_my_task
```

The next example is a bit more sophisticated, and tests whether the **my_service** variable has a value. If it does, the value of **my_service** is used as the name of the package to install. If the **my_service** variable is not defined, then the task is skipped without an error.

```
---
- hosts: all
  vars:
    my_service: httpd

  tasks:
    - name: "{{ my_service }} package is installed"
      yum:
        name: "{{ my_service }}"
      when: my_service is defined
```

The following table shows some of the operators that administrators can use when working with conditionals:

Example Conditionals

Operator	Example
Equal (value is a string)	ansible_machine == "x86_64"
Equal (value is numeric)	max_memory == 512
Less than	min_memory < 128
Greater than	min_memory > 256
Less than or equal to	min_memory <= 256
Greater than or equal to	min_memory >= 512
Not equal to	min_memory != 512
Variable exists	min_memory is defined

Operator	Example
Variable does not exist	<code>min_memory is not defined</code>
Variable is set to 1 , True , or yes	<code>available_memory</code>
Variable is set to 0 , False , or no	<code>not available_memory</code>
First variable's value is present as a value in second variable's list	<code>my_special_user in superusers</code>

The last entry in the preceding table might be a bit confusing at first. In the example for that entry, `my_special_user` is a variable which has some value. The variable `superusers` is a variable which has a list for a value. If the value of `my_special_user` is in the `superusers` list, the conditional passes and the task runs.

The following example will use this type of conditional. Given the variable settings and the conditional shown in the example, the task will run:

```
---
- hosts: all
  vars:
    my_special_user: devops

    superusers:
      - root
      - devops
      - toor

    tasks:
      - name: Task runs if my_special_user is in superusers
        user:
          name: "{{ my_special_user }}"
          groups: wheel
          append: yes
        when: my_special_user in superusers
```

Here is another example using the same kind of conditional. It uses two *magic variables* that Ansible automatically sets. The task will only run if the value of the managed host's `inventory_hostname` variable (containing the name of the managed host from the inventory file) is listed as a member of the host group **databases**.

```
- name: Create the database admin
  user:
    name: db_admin
  when: inventory_hostname in groups["databases"]
```

(Magic variables are discussed in more depth in the Ansible documentation at http://docs.ansible.com/ansible/playbooks_variables.html#magic-variables-and-how-to-access-information-about-other-hosts.)



Important

Notice the indentation of the **when** statement. Because the **when** statement is not a module variable, it must be placed "outside" the module by being indented at the top level of the task.

A task is a YAML hash/dictionary, and the **when** statement is simply one more key in the task like the task's name and the module it uses. A common convention places any **when** directive that might be present after the task's name and the module (and module arguments).

Testing Multiple Conditions

One **when** statement can be used to evaluate multiple values. To do so, conditionals can be combined with the **and** and **or** keywords or grouped with parentheses.

The following snippets show some examples of how to express multiple conditions.

- With the **and** operation, both conditions have to be true for the entire conditional statement to be met. For example, the following condition will be met if the installed kernel is the specified version and if the **inventory_hostname** is in the **staging** group:

```
ansible_kernel == 3.10.0-327.el7.x86_64 and inventory_hostname in groups['staging']
```

- If a conditional statement should be met when either condition is true, then the **or** statement should be used. For example, the following condition is met if the machine is running either Red Hat Enterprise Linux or Fedora:

```
ansible_distribution == "RedHat" or ansible_distribution == "Fedora"
```

- More complex conditional statements can be clearly expressed through grouping conditions with parentheses to ensure that they are correctly interpreted. For example, the following conditional statement is met if the machine is running either Red Hat Enterprise Linux 7 or Fedora 23:

```
(ansible_distribution == "RedHat" and ansible_distribution_major_version == 7) or  
(ansible_distribution == "Fedora" and ansible_distribution_major_version == 23)
```

Combining Loops and Conditional Tasks

Loops and conditionals can be combined. In the following example, the *mariadb-server* package will be installed by the **yum** module if there is a file system mounted on / with more than 300 MB free. The **ansible_mounts** fact is a list of dictionaries, each one representing facts about one mounted file system. The loop iterates over each dictionary in the list, and the conditional statement is not met unless a dictionary is found representing a mounted file system where both conditions are true.

```
- name: install mariadb-server if enough space on root
  yum:
    name: mariadb-server
    state: latest
```

```
with_items: "{{ ansible_mounts }}"
when: item.mount == "/" and item.size_available > 3000000000
```



Important

When combining **when** with **with_items**, be aware that the **when** statement is processed for each item.

Here is another example combining conditionals and registered variables. The following annotated playbook will restart the **httpd** service only if the **postfix** service is running.

```
- hosts: all
  tasks:
    - name: Postfix server status
      command: /usr/bin/systemctl is-active postfix ①
      ignore_errors: yes ②
      register: result ③

    - name: Restart Apache HTTPD if Postfix running
      service:
        name: httpd
        state: restarted
      when: result.rc == 0 ④
```

- ①** Is Postfix running or not?
- ②** If it is not running and the command "fails", do not stop processing
- ③** Saves information on the module's result in a variable named **result**
- ④** Evaluates the output of the Postfix task. If the exit code of the **systemctl** command is 0, then Postfix is active and this task will restart the **httpd** service.



References

Loops – Ansible Documentation
http://docs.ansible.com/ansible/playbooks_loops.html

Conditionals – Ansible Documentation
http://docs.ansible.com/ansible/playbooks_conditionals.html

What Makes A Valid Variable Name – Variables – Ansible Documentation
http://docs.ansible.com/ansible/playbooks_variables.html#what-makes-a-valid-variable-name

Guided Exercise: Constructing Flow Control

In this exercise, you will construct conditionals and loops in Ansible playbooks

Outcomes

You should be able to:

- Implement Ansible conditionals using the **when** statement.
- Use Ansible **with_items** loops in conjunction with conditionals.

Before you begin

From **workstation**, run the lab setup script to confirm the environment is ready for the lab to begin. The script creates the working directory, called **dev-flowcontrol**, and populates it with an Ansible configuration file and host inventory.

```
[student@workstation ~]$ lab task-control-flowcontrol setup
```

Steps

1. As the **student** user on **workstation**, change to the directory, **/home/student/dev-flowcontrol**.

```
[student@workstation ~]$ cd ~/dev-flowcontrol
[student@workstation dev-flowcontrol]$
```

2. Create a task file named **configure_database.yml**. This will define the tasks to install the extra packages, update **/etc/my.cnf** from a copy stored on a web site, and start **mariadb** on the managed hosts. The include file can and will use the variables you defined in the **playbook.yml** file and inventory. The **get_url** module will need to set **force=yes** so that the **my.cnf** file is updated even if it already exists on the managed host, and will need to set correct permissions as well as SELinux contexts on the **/etc/my.cnf** file. When you are finished, save the file and exit the editor. The file should read as follows:

```
- yum:
  name: "{{ extra_packages }}"

- get_url:
  url: "http://materials.example.com/task_control/my.cnf"
  dest: "{{ configure_database_path }}"
  owner: mysql
  group: mysql
  mode: 0644
  seuser: system_u
  setype: mysqld_etc_t
  force: yes

- service:
  name: "{{ db_service }}"
  state: started
  enabled: true
```

3. In the same directory, create the **playbook.yml** playbook. Define a list variable, **db_users**, that consists of a list of two users, **db_admin** and **db_user**. Add a **configure_database_path** variable set to the file **/etc/my.cnf**.

Create a task that uses a loop to create the users only if the managed host belongs to the databases host group. The file should read as follows:

```
---
- hosts: all
  vars:
    db_package: mariadb-server
    db_service: mariadb
    db_users:
      - db_admin
      - db_user
    configure_database_path: /etc/my.cnf

  tasks:
    - name: Create the MariaDB users
      user:
        name: "{{ item }}"
      with_items: "{{ db_users }}"
      when: inventory_hostname in groups['databases']
```

4. In the playbook, add a task that uses the **db_package** variable to install the database software, only if the variable has been defined. The task should read as follows:

```
- name: Install the database server
  yum:
    name: "{{ db_package }}"
  when: db_package is defined
```

5. In the playbook, create a task to do basic configuration of the database. The task will run only when **configure_database_path** is defined. This task should include the **configure_database.yml** task file and define a local array, **extra_packages**, which will be used to specify additional packages needed for this configuration. Set that list variable to include a list of three packages: *mariadb-bench*, *mariadb-libs*, and *mariadb-test*. When done, save the playbook and exit the editor.

```
- name: Configure the database software
  include: configure_database.yml
  vars:
    extra_packages:
      - mariadb-bench
      - mariadb-libs
      - mariadb-test
  when: configure_database_path is defined
```

6. Check the final **playbook.yml** file before running it. It should now read in its entirety:

```
---
- hosts: all
  vars:
    db_package: mariadb-server
    db_service: mariadb
    db_users:
```

```

- db_admin
- db_user
configure_database_path: /etc/my.cnf

tasks:
- name: Create the MariaDB users
  user:
    name: "{{ item }}"
  with_items: "{{ db_users }}"
  when: inventory_hostname in groups['databases']

- name: Install the database server
  yum:
    name: "{{ db_package }}"
  when: db_package is defined

- name: Configure the database software
  include: configure_database.yml
  vars:
    extra_packages:
      - mariadb-bench
      - mariadb-libs
      - mariadb-test
  when: configure_database_path is defined

```

- Before running the playbook, verify its syntax is correct by running **ansible-playbook --syntax-check**. If it reports any errors, correct them before moving to the next step. You should see output similar to the following:

```
[student@workstation dev-flowcontrol]$ ansible-playbook --syntax-check playbook.yml
playbook: playbook.yml
```

- Run the playbook to install and configure the database on the managed hosts.

```
[student@workstation dev-flowcontrol]$ ansible-playbook playbook.yml
PLAY [all] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]
... Output omitted ...
TASK [yum] ****
changed: [servera.lab.example.com]
... Output omitted ...
```

The output confirms the task file has been successfully included and executed.

- Manually verify that the necessary packages have been installed on **servera**, that the **/etc/my.cnf** file is in place with the correct permissions, and that the two users have been created.
- Use an ad hoc command from **workstation** to **servera** to confirm the packages have been installed.

```
[student@workstation dev-flowcontrol]$ ansible all -a 'yum list installed  
mariadb-bench mariadb-libs mariadb-test'  
... Output omitted ...  
  
servera.lab.example.com | SUCCESS | rc=0 >>  
Loaded plugins: langpacks, search-disabled-repos  
Installed Packages  
mariadb-bench.x86_64           1:5.5.52-1.el7          @rhel_dvd  
mariadb-libs.x86_64            1:5.5.52-1.el7          installed  
mariadb-test.x86_64           1:5.5.52-1.el7          @rhel_dvd
```

9.2. Confirm the **my.cnf** file has been successfully copied under **/etc/**.

```
[student@workstation dev-flowcontrol]$ ansible all -a 'grep Ansible /etc/my.cnf'  
servera.lab.example.com | SUCCESS | rc=0 >>  
# Ansible file
```

9.3. Confirm the two users have been created.

```
[student@workstation dev-flowcontrol]$ ansible all -a 'id db_user'  
servera.lab.example.com | SUCCESS | rc=0 >>  
uid=1003(db_user) gid=1003(db_user) groups=1003(db_user)  
[student@workstation dev-flowcontrol]$ ansible all -a 'id db_admin'  
servera.lab.example.com | SUCCESS | rc=0 >>  
uid=1002(db_admin) gid=1002(db_admin) groups=1002(db_admin)
```

Evaluation

Run the **lab task-control-flowcontrol grade** command from **workstation** to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab task-control-flowcontrol grade
```

Cleanup

Run the **lab task-control-flowcontrol cleanup** command to cleanup after the lab.

```
[student@workstation ~]$ lab task-control-flowcontrol cleanup
```

Implementing Handlers

Objectives

After completing this section, students should be able to:

- Implement handlers in playbooks

Ansible Handlers

Ansible modules are designed to be *idempotent*. This means that in a properly written playbook, the playbook and its tasks can be run multiple times without changing the managed host, unless they need to make a change in order to get the managed host to the desired state.

However, sometimes when a task does make a change to the system, a further task may need to be run. For example, a change to a service's configuration file may then require that the service be reloaded so that the changed configuration takes effect.

Handlers are tasks that respond to a notification triggered by other tasks. Each handler has a globally-unique name, and is triggered at the end of a block of tasks in a playbook. If no task notifies the handler by name, it will not run. If one or more tasks notify the handler, it will run exactly once after all other tasks in the play have completed. Because handlers are tasks, administrators can use the same modules in handlers that they would for any other task. Normally, handlers are used to reboot hosts and restart services.

Handlers can be seen as *inactive* tasks that only get triggered when explicitly invoked using a **notify** statement. The following snippet shows how the Apache server is only restarted by the **restart_apache** handler when a configuration file is updated and notifies it:

```
tasks:
  - name: copy demo.example.conf configuration template①
    copy:
      src: /var/lib/templates/demo.example.conf.template
      dest: /etc/httpd/conf.d/demo.example.conf
    notify: ②
      - restart_apache③

handlers: ④
  - name: restart_apache⑤
    service: ⑥
      name: httpd
      state: restarted
```

- ① The task that notifies the handler.
- ② The **notify** statement indicates the task needs to trigger a handler.
- ③ The name of the handler to run.
- ④ The statement starts the handlers section.
- ⑤ The name of the handler invoked by tasks.
- ⑥ The module to use for the handler.

In the previous example, the **restart_apache** handler will trigger when notified by the **copy** task that a change happened. A task may call more than one handler in their **notify** section. Ansible treats the **notify** statement as an array and iterates over the handler names:

```
tasks:
  - name: copy demo.example.conf configuration template
    copy:
      src: /var/lib/templates/demo.example.conf.template
      dest: /etc/httpd/conf.d/demo.example.conf
    notify:
      - restart_mysql
      - restart_apache

handlers:
  - name: restart_mysql
    service:
      name: mariadb
      state: restarted

  - name: restart_apache
    service:
      name: httpd
      state: restarted
```

Using Handlers

As discussed in the Ansible documentation, there are some important things to remember about using handlers:

- Handlers are always run in the order in which the **handlers** section is written in the play, not in the order in which they are listed by the **notify** statement on a particular task.
- Handlers run after all other tasks in the play complete. A handler called by a task in the **tasks:** part of the playbook will not run until *all* of the tasks under **tasks:** have been processed.
- Handler names live in a global namespace. If two handlers are incorrectly given the same name, only one will run.
- Handlers defined inside an **include** can not be notified.
- Even if more than one task notifies a handler, the handler will only run once. If no tasks notify it, a handler will not run.
- If a task that includes a **notify** does not execute (for example, a package is already installed), the handler will not be notified. The handler will be skipped unless another task notifies it. Ansible notifies handlers only if the task acquires the **CHANGED** status.



Important

Handlers are meant to perform an action upon the execution of a task; they should not be used as a replacement for tasks.



References

Intro to Playbooks – Ansible Documentation
http://docs.ansible.com/ansible/playbooks_intro.html

Guided Exercise: Implementing Handlers

In this exercise, you will implement handlers in playbooks.

Outcomes

You should be able to:

- Define handlers in playbooks and notify them for configuration change.

Before you begin

Run **lab task-control-handlers setup** from **workstation** to configure the environment for the exercise. The script creates the **dev-handlers** project directory as well as the Ansible configuration file and the host inventory file.

```
[student@workstation ~]$ lab task-control-handlers setup
```

Steps

1. From **workstation.lab.example.com**, open a new terminal and change to the **dev-handlers** project directory.

```
[student@workstation ~]$ cd ~/dev-handlers
[student@workstation dev-handlers]$
```

2. In that directory, use a text editor to create the **configure_db.yml** playbook file. This file

will install a database server and create some users; when the database server is installed, the playbook restarts the service.

- 2.1. Start the playbook with the initialization of some variables: **db_packages**, which defines the name of the packages to install for the database service; **db_service**, which defines the name of the database service; **src_file** for the URL of the configuration file to install; and **dst_file** for the location of the installed configuration file on the managed hosts. The playbook should read as follows:

```
---
- name: Installing Mariadb server
  hosts: databases
  vars:
    db_packages:
      - mariadb-server
      - MySQL-python
    db_service: mariadb
    src_file: "http://materials.example.com/task_control/my.cnf.template"
    dst_file: /etc/my.cnf
```

- 2.2. In the **configure_db.yml** file, define a task that uses the **yum** module to install the required database packages as defined by the **db_packages** variable. Notify the handler, **start_service**, in order to start the service. The task should read as follows:

```
tasks:
  - name: Install {{ db_packages }} package
    yum:
      name: "{{ item }}"

```

```

    state: latest
    with_items: "{{ db_packages }}"
    notify:
      - start_service

```

- 2.3. Add a task to download **my.cnf.template** to **/etc/my.cnf** on the managed host, using the **get_url** module. Add a condition that notifies the handler, **restart_service**, as well as **set_password**, to restart the database service and set the administrative password. The task should read:

```

- name: Download and install {{ dst_file }}
  get_url:
    url: "{{ src_file }}"
    dest: "{{ dst_file }}"
    owner: mysql
    group: mysql
    force: yes
  notify:
    - restart_service
    - set_password

```

- 2.4. Define the three handlers the tasks needs. The **start_service** handle will start the **mariadb** service; the **restart_service** handler will restart the **mariadb** service; and the **set_password** handler will set the administrative password for the database service.

Define the **start_service** handler. It should read as follows:

```

handlers:
  - name: start_service
    service:
      name: "{{ db_service }}"
      state: started

```

- 2.5. Define the second handler, **restart_service**. It should read as follows:

```

- name: restart_service
  service:
    name: "{{ db_service }}"
    state: restarted

```

- 2.6. Finally, define the handler that sets the administrative password. The handler will use the **mysql_user** module to perform the command. The handler should read as follows:

```

- name: set_password
  mysql_user:
    name: root
    password: redhat

```

- 2.7. When completed, the playbook should look like the following:

```

---
- name: Installing Mariadb server

```

```
hosts: databases
vars:
  db_packages:
    - mariadb-server
    - MySQL-python
  db_service: mariadb
  src_file: "http://materials.example.com/task_control/my.cnf.template"
  dst_file: /etc/my.cnf

tasks:
  - name: Install {{ db_packages }} package
    yum:
      name: "{{ item }}"
      state: latest
    with_items: "{{ db_packages }}"
    notify:
      - start_service
  - name: Download and install {{ dst_file }}
    get_url:
      url: "{{ src_file }}"
      dest: "{{ dst_file }}"
      owner: mysql
      group: mysql
      force: yes
    notify:
      - restart_service
      - set_password

handlers:
  - name: start_service
    service:
      name: "{{ db_service }}"
      state: started

  - name: restart_service
    service:
      name: "{{ db_service }}"
      state: restarted

  - name: set_password
    mysql_user:
      name: root
      password: redhat
```

3. Before running the playbook, verify its syntax is correct by running **ansible-playbook --syntax-check**. If it reports any errors, correct them before moving to the next step. You should see output similar to the following:

```
[student@workstation dev-handlers]$ ansible-playbook --syntax-check configure_db.yml
playbook: configure_db.yml
```

4. Run the **configure_db.yml** playbook. Notice the output shows the handlers are being executed.

```
[student@workstation dev-handlers]# ansible-playbook configure_db.yml
PLAY [Installing Mariadb server]*****
... Output omitted ...
```

```
RUNNING HANDLER [start_service] ****
changed: [servera.lab.example.com]

RUNNING HANDLER [restart_service] ****
changed: [servera.lab.example.com]

RUNNING HANDLER [set_password] ****
changed: [servera.lab.example.com]
```

- Run the playbook again. This time the handlers are skipped.

```
[student@workstation dev-handlers]# ansible-playbook configure_db.yml

PLAY [Installing Mariadb server]*****
... Output omitted ...

PLAY RECAP ****
servera.lab.example.com : ok=3    changed=0    unreachable=0    failed=0
```

- Because handlers are executed once, they can help prevent failures. Update the playbook to add a task after installing `/etc/my.cnf` that sets the MySQL admin password like the `set_password` handler. This will show you why using a handler in this situation is better than a simple task. The task should read as follows:

```
- name: Set the MySQL password
  mysql_user:
    name: root
    password: redhat
```

- Run the playbook again. The task should fail since the MySQL password has already been set.

```
[student@workstation dev-handlers]# ansible-playbook configure_db.yml

TASK [Set the MySQL password] ****
fatal: [servera.lab.example.com]: FAILED! => {"changed": false, "failed": true,
"msg": "unable to connect to database, check login_user and login_password are
correct or /root/.my.cnf has the credentials. Exception message: (1045, \"Access
denied for user 'root'@'localhost' (using password: NO)\")"
      to retry, use: --limit @/home/student/dev-handlers/configure_db.retry

PLAY RECAP ****
servera.lab.example.com : ok=3    changed=0    unreachable=0    failed=1
```

Evaluation

From workstation, run the `lab task-control-handlers grade` command to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab task-control-handlers grade
```

Cleanup

Run the `lab task-control-handlers cleanup` command to clean up after completing the lab.

```
[student@workstation ~]$ lab task-control-handlers cleanup
```

Implementing Tags

Objectives

After completing this section, students should be able to:

- Set tags to control which parts of a playbook should be run.

Tagging Ansible Resources

Sometimes it is useful to be able to run subsets of the tasks in a playbook. *Tags* can be applied to specific resources as a text label in order to allow this. Tagging a resource only requires that the **tags** keyword be used, followed by a list of tags to apply. When plays are tagged, the **--tags** option can be used with **ansible-playbook** to filter the playbook to only execute specific tagged plays. Tags are available for the following resources:

- In playbooks, each task can be tagged, using the **tags** keyword:

```
tasks:
  - name: Package {{ item }} is installed
    yum:
      name: "{{ item }}"
      state: installed
    with_items:
      - postfix
      - mariadb-server
    tags:
      - packages
```

- When a task file is included in a playbook, the task can be tagged, allowing administrators to set a global tag for the **include** statement:

```
- include: common.yml
tags:
  - webproxy
  - webserver
```



Note

When tagging a role or an **include** statement, all tasks they define are also tagged.

```
roles:
  - { role: databases, tags: ['production', 'staging'] }
```



Important

When roles or **include** statements are tagged, the tag is not a way to exclude some of the tagged tasks the included files contain. Tags in this context are a way to apply a global tag to all tasks.

Managing Tagged Resources

When a tag has been applied to a resource, the **ansible-playbook** command can be used with the **--tags** or **--skip-tags** argument to either execute the tagged resources, or prevent the tagged resources from being included in the play. The following playbook contains two tasks; the first one is tagged with the **webserver** tag, the other one does not have any tag associated with it:

```
---
- name: Example play using tagging
  hosts:
    - servera.lab.example.com
    - serverb.lab.example.com

  tasks:
    - name: httpd is installed
      yum:
        name: httpd
        state: latest
      tags: webserver

    - name: postfix is installed
      yum:
        name: postfix
        state: latest
```

To only run the first task, the **--tags** argument can be used:

```
[user@demo ~]$ ansible-playbook main.yml --tags 'webserver'

PLAY [Example play using tagging] ****

TASK [Gathering Facts] ****
ok: [serverb.lab.example.com]
ok: [servera.lab.example.com]

TASK [httpd is installed] ****
ok: [servera.lab.example.com]
ok: [serverb.lab.example.com]

PLAY RECAP ****
servera.lab.example.com    : ok=2    changed=0    unreachable=0    failed=0
serverb.lab.example.com    : ok=2    changed=0    unreachable=0    failed=0
```

Because the **--tags** option was specified, the playbook only ran the task tagged with the **webserver** tag. To skip tasks with a specific tag and only run the tasks without that tag, the **--skip-tags** option can be used:

```
[user@demo ~]$ ansible-playbook main.yml --skip-tags 'webserver'

PLAY [Example play using tagging] ****

TASK [Gathering Facts] ****
ok: [serverb.lab.example.com]
ok: [servera.lab.example.com]

TASK [postfix is installed] ****
ok: [serverb.lab.example.com]
ok: [servera.lab.example.com]
```

```
PLAY RECAP ****
servera.lab.example.com : ok=2    changed=0    unreachable=0    failed=0
serverb.lab.example.com : ok=2    changed=0    unreachable=0    failed=0
```

Special Tags

Ansible has a special tag that can be assigned in a playbook: **always**. This tag causes the task to always be executed even if the **--skip-tags** option is used, unless explicitly skipped with **--skip-tags always**.

There are three special tags that can be used from the command-line with the **--tags** option:

1. The **tagged** keyword is used to run any tagged resource.
2. The **untagged** keyword does the opposite of the **tagged** keyword by excluding all tagged resources from the play.
3. The **all** keyword allows administrators to include all tasks in the play. This is the default behavior of the command line.

Demonstration: Implementing Tags

1. From **workstation.lab.example.com**, open a new terminal and change to the **demo-tags** project directory. Create the **prepare_db.yml** task file and define a task that installs the required services for the database. Tag the task as **dev**, and have it notify the **start_db** handler. The file should read as follows:

```
---
- name: Install database packages
  yum:
    name: "{{ item }}"
    state: latest
  with_items: "{{ db_packages }}"
  tags:
    - dev
  notify:
    - start_db
```

2. Define a second task to retrieve the database configuration file if the **dev** tag is active in the execution environment, and that restarts that database service. The task will use a conditional to ensure the configuration file path is defined. The task should read as follows:

```
- name: Get small config file
  get_url:
    url: "{{ db_config_src_path_small }}"
    dest: "{{ db_config_path }}"
  when: db_config_src_path_small is defined
  notify:
    - restart_db
  tags:
    - dev
```

3. Define a third task to retrieve a different database configuration file if the **prod** tag is active in the execution environment, and that restarts the database service. Like the previous task, this task will use a conditional to ensure that the configuration file path is defined. The task should read as follows:

```
- name: Get large config file
get_url:
  url: "{{ db_config_src_path_large }}"
  dest: "{{ db_config_path }}"
when: db_config_src_path_large is defined
notify:
  - restart_db
tags:
  - prod
```

4. Define a task that sets the Message of The Day for the managed host. Tag the task with the **dev** task. The task should read as follows:

```
- name: Update motd for development
copy:
  content: "This is a development server"
  dest: /etc/motd
tags:
  - dev
```

5. Repeat the previous step but change both the tag as well as the command. The task should read as follows:

```
- name: Update motd for production
copy:
  content: "This is a production server"
  dest: /etc/motd
tags:
  - prod
```

6. When completed, the task file should read as follows:

```
---
- name: Install database packages
yum:
  name: "{{ item }}"
  state: latest
with_items: "{{ db_packages }}"
tags:
  - dev
notify:
  - start_db

- name: Get small config file
get_url:
  url: "{{ db_config_src_path_small }}"
  dest: "{{ db_config_path }}"
when: db_config_src_path_small is defined
notify:
  - restart_db
tags:
  - dev

- name: Get large config file
get_url:
  url: "{{ db_config_src_path_large }}"
  dest: "{{ db_config_path }}"
```

```

when: db_config_src_path_large is defined
notify:
  - restart_db
tags:
  - prod

- name: Update motd for development
copy:
  content: "This is a development server"
  dest: /etc/motd
tags:
  - dev

- name: Update motd for production
copy:
  content: "This is a production server"
  dest: /etc/motd
tags:
  - prod

```

- In the project directory, create the main playbook, **playbook.yml** for servers in the **databases** group. The playbook will define the variables required by the task file upon import. The playbook should read as follows:

```

---
- hosts: all
vars:
  db_packages:
    - mariadb-server
    - MySQL-python
  db_config_url: http://materials.example.com/task_control
  db_config_src_path_small: "{{ db_config_url }}/my.cnf.small"
  db_config_src_path_large: "{{ db_config_url }}/my.cnf.large"
  db_config_path: /etc/my.cnf
  db_service: mariadb

```

- Define the first task that includes the task file; the task file will use a conditional to ensure the managed host is in the group **databases**. The task should read as follows:

```

tasks:
- include:
  prepare_db.yml
when: inventory_hostname in groups['databases']

```

- Define the two handlers the task file requires, **start_db** and **restart_db**. The **handlers** block should read as follows:

```

handlers:
- name: start_db
  service:
    name: "{{ db_service }}"
    state: started

- name: restart_db
  service:
    name: "{{ db_service }}"
    state: restarted

```

10. When completed, the playbook should read as follows:

```
---
- hosts: all
  vars:
    db_packages:
      - mariadb-server
      - MySQL-python
    db_config_url: http://materials.example.com/task_control
    db_config_src_path_small: "{{ db_config_url }}/my.cnf.small"
    db_config_src_path_large: "{{ db_config_url }}/my.cnf.large"
    db_config_path: /etc/my.cnf
    db_service: mariadb

  tasks:
    - include:
        prepare_db.yml
      when: inventory_hostname in groups['databases']

  handlers:
    - name: start_db
      service:
        name: "{{ db_service }}"
        state: started

    - name: restart_db
      service:
        name: "{{ db_service }}"
        state: restarted
```

11. Run the playbook, applying the **dev** tag using the **--tags** option.

```
[student@workstation demo-tags]$ ansible-playbook playbook.yml --tags 'dev'
... Output omitted ...
TASK [Get small config file] ****
changed: [servera.lab.example.com]

TASK [Update motd for development] ****
changed: [servera.lab.example.com]
... Output omitted ...
```

Notice the configuration file that has been retrieved (**my.cnf.small**).

12. Run an ad hoc command to display the **/etc/motd** file on **servera**.

```
[student@workstation dev-tags]$ ansible databases -a 'cat /etc/motd'
servera.lab.example.com | SUCCESS | rc=0 >>
This is a development server
```

13. Run the playbook again, this time skipping the tasks tagged as **dev**:

```
[student@workstation demo-tags]$ ansible-playbook playbook.yml --skip-tags 'dev'
... Output omitted ...
TASK [Get large config file] ****
ok: [servera.lab.example.com]

TASK [Update motd for production] ****
changed: [servera.lab.example.com]
```

```
... Output omitted ...
```

14. Run an ad hoc command to display the **/etc/motd** file on **servera**.

```
[student@workstation dev-tags]$ ansible databases -a 'cat /etc/motd'  
servera.lab.example.com | SUCCESS | rc=0 >>  
This is a production server
```

References



Tags – Ansible Documentation

http://docs.ansible.com/ansible/playbooks_tags.html

Task And Handler Organization For A Role – Best Practices – Ansible Documentation

http://docs.ansible.com/ansible/playbooks_best_practices.html#task-and-handler-organization-for-a-role

Guided Exercise: Implementing Tags

In this exercise, you will implement tags in a playbook and run the playbook.

Outcomes

You should be able to:

- Tag Ansible tasks.
- Filter tasks based on tags when running playbooks.

Before you begin

Run the **lab task-control-tags setup** command from **workstation** to prepare the environment for this exercise. The script creates the working directory, **dev-tags**, and populates it with an Ansible configuration file and host inventory.

```
[student@workstation ~]$ lab task-control-tags setup
```

Steps

1. Change to the **dev-tags** project directory that the lab script created.

```
[student@workstation ~]$ cd ~/dev-tags  
[student@workstation dev-tags]$
```

2. The following steps will edit the same **configure_mail.yml** task file.

In the project directory, create the **configure_mail.yml** task file. The task file contains instructions to install the required packages for the mail server, as well as instructions to retrieve the configuration files for the mail server.

- 2.1. Create the first task that uses the **yum** module to install the *postfix* package. Notify the **start_postfix** handler, and tag the task as **server** using the **tags** keyword. The task should read as follows:

```
---  
- name: Install postfix  
  yum:  
    name: postfix  
    state: latest  
  tags:  
    - server  
  notify:  
    - start_postfix
```

- 2.2. Add a task that installs the *dovecot* package using the **yum** module. It should notify the **start_dovecot** handler. Tag the task as **client**. The task should read as follows:

```
- name: Install dovecot  
  yum:  
    name: dovecot  
    state: latest  
  tags:  
    - client
```

```
    notify:
      - start_dovecot
```

- 2.3. Define a task that uses the **get_url** module to retrieve the Postfix configuration file. Notify the **restart_postfix** handler and tag the task as **server**. The task should read as follows:

```
- name: Download main.cf configuration
  get_url:
    url: http://materials.example.com/task_control/main.cf
    dest: /etc/postfix/main.cf
  tags:
    - server
  notify:
    - restart_postfix
```

- 2.4. When completed, the task file should read as follows:

```
---
- name: Install postfix
  yum:
    name: postfix
    state: latest
  tags:
    - server
  notify:
    - start_postfix

- name: Install dovecot
  yum:
    name: dovecot
    state: latest
  tags:
    - client
  notify:
    - start_dovecot

- name: Download main.cf configuration
  get_url:
    url: http://materials.example.com/task_control/main.cf
    dest: /etc/postfix/main.cf
  tags:
    - server
  notify:
    - restart_postfix
```

3. The following steps will edit the same **playbook.yml** playbook file.

- 3.1. Create a playbook file named **playbook.yml**. Define the playbook for all hosts. The playbook should read as follows:

```
---
- hosts: all
```

- 3.2. Define the task that includes the task file **configure_mail.yml** using the **include** module. Add a conditional to only run the task for the hosts in the **mailservers** group. The task should read as follows:

```
tasks:
  - name: Include configure_mail.yml
    include:
      configure_mail.yml
    when: inventory_hostname in groups['mailservers']
```

- 3.3. Define the three handlers the task file requires: **start_postfix**, **start_dovecot**, and **restart_postfix**.

Define **start_postfix** handler to start the mail server. The handler should read as follows:

```
handlers:
  - name: start_postfix
    service:
      name: postfix
      state: started
```

- 3.4. Define the **start_dovecot** handler to start the mail client. The handler should read as follows:

```
  - name: start_dovecot
    service:
      name: dovecot
      state: started
```

- 3.5. Define the **restart_postfix** handler that restarts the mail server. The handler should read as follows:

```
  - name: restart_postfix
    service:
      name: postfix
      state: restarted
```

- 3.6. When completed, the playbook should read as follows:

```
---
- hosts: all

  tasks:
    - name: Include configure_mail.yml
      include:
        configure_mail.yml
      when: inventory_hostname in groups['mailservers']

  handlers:
    - name: start_postfix
      service:
        name: postfix
        state: started

    - name: start_dovecot
      service:
        name: dovecot
        state: started
```

```
- name: restart_postfix
  service:
    name: postfix
    state: restarted
```

4. Before running the playbook, verify its syntax is correct by running **ansible-playbook --syntax-check**. If it reports any errors, correct them before moving to the next step. You should see output similar to the following:

```
[student@workstation dev-tags]$ ansible-playbook --syntax-check playbook.yml
playbook: playbook.yml
```

5. Run the playbook, only applying the **server** tagged tasks using the **--tags** option. Notice how only the **start_postfix** handler gets triggered.

```
[student@workstation dev-tags]$ ansible-playbook playbook.yml --tags 'server'
... Output omitted ...
RUNNING HANDLER [start_postfix] ****
changed: [servera.lab.example.com]
... Output omitted ...
```

6. Run an ad hoc command to ensure the *postfix* package has been successfully installed.

```
[student@workstation dev-tags]$ ansible mailservers -a 'yum list installed postfix'
servera.lab.example.com | SUCCESS | rc=0 >>
Loaded plugins: langpacks, search-disabled-repos
Installed Packages
postfix.x86_64                  2:2.10.1-6.el7          @rhel_dvd
```

7. Run the playbook again, but this time skip the tasks tagged with the **server** tag. The play will install the *dovecot* package, because the task is tagged with the **client** tag, and it will trigger the **start_dovecot** handler.

```
[student@workstation dev-tags]$ ansible-playbook playbook.yml --skip-tags 'server'
... Output omitted ...
TASK [Install dovecot] ****
changed: [servera.lab.example.com]

RUNNING HANDLER [start_dovecot] ****
changed: [servera.lab.example.com]
... Output omitted ...
```

8. Run an ad hoc command to ensure the *dovecot* package has been successfully installed.

```
[student@workstation dev-tags]$ ansible mailservers -a 'yum list installed dovecot'
servera.lab.example.com | SUCCESS | rc=0 >>
Loaded plugins: langpacks, search-disabled-repos
Installed Packages
dovecot.x86_64                  1:2.2.10-7.el7          @rhel_dvd
```

Evaluation

Run the **lab task-control-tags grade** command from **workstation** to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab task-control-tags grade
```

Cleanup

Run the **lab task-control-tags cleanup** command to cleanup after the lab.

```
[student@workstation ~]$ lab task-control-tags cleanup
```

Handling Errors

Objectives

After completing this section, students should be able to:

- Resolve errors in a playbook.

Errors in Plays

Ansible evaluates the return code of each task to determine whether the task succeeded or failed. Normally, when a task fails Ansible immediately aborts the rest of the play on that host, skipping all subsequent tasks.

However, sometimes you may want to have play execution continue even if a task fails. For example, you might expect that a particular task could fail, and you might want to recover by running some other task conditionally. There are a number of Ansible features that can be used to manage task errors.

Ignoring Task Failure

By default, if a task fails, the play is aborted. However, this behavior can be overridden by ignoring failed tasks. To do so, the **ignore_errors** keyword needs to be used in a task.

The following snippet shows how to use **ignore_errors** on a task to continue playbook execution on the host even if the task fails. For example, if the *notapkg* package does not exist the yum module will fail, but having **ignore_errors** set to **yes** will allow execution to continue.

```
- yum:
  name: notapkg
  state: latest
  ignore_errors: yes
```

Forcing Execution of Handlers after Task Failure

Normally when a task fails and the play aborts on that host, any handlers which had been notified by earlier tasks in the play will not run. If you set the **force_handlers: yes** directive on the play, then notified handlers will be called even if the play aborted because a later task failed.

The following snippet shows how to use the **force_handlers** keyword in a play to forcefully execute the handler even if a task fails:

```
---
- hosts: all
  force_handlers: yes
  tasks:
    - name: a task which always notifies its handler
      command: /bin/true
      notify: restart the database

    - name: a task which fails because the package doesn't exist
      yum:
        name: notapkg
        state: latest
```

```
handlers:
  - name: restart the database
    service:
      name: mariadb
      state: restarted
```

Note

Remember that handlers are notified when a task reports a "changed" result but are not notified when it reports an "ok" or "failed" result.

Specifying Task Failure Conditions

You can use the **failed_when** directive on a task to specify which conditions indicate that the task has failed. This is often used with "run command" modules that may successfully execute a command, but the command's output or exit code may indicate a failure.

For example, you can run a script that outputs an error message and use that message to define the failed state for the task. The following snippet shows how the **failed_when** keyword can be used in a task:

```
tasks:
  - shell: /usr/local/bin/create_users.sh
    register: command_result
    failed_when: "'Password missing' in command_result.stdout"
```

Specifying When a Task Reports "Changed" Results

When a task makes a change to a managed host, it reports the **changed** state and notifies handlers. When a task does not need to make a change, it reports **ok** and does not notify handlers.

The **changed_when** directive can be used to control when a task reports that it has changed. For example, the **shell** module in the next example is being used to get a Kerberos credential which will be used by subsequent tasks. It normally would always report "changed" when it runs. To suppress that change, **changed_when: false** is set so that it only reports "ok" or "failed".

```
- name: get Kerberos credentials as "admin"
  shell: echo "{{ krb_admin_pass }}" | kinit -f admin
  changed_when: false
```

Here is another example using the **shell** module that reports "changed" based on output of the module that is collected by a registered variable:

```
tasks:
  - shell:
    cmd: /usr/local/bin/upgrade-database
    register: command_result
    changed_when: "'Success' in command_result.stdout"
    notify:
      - restart_database

handlers:
```

```
- name: restart_database
  service:
    name: mariadb
    state: restarted
```

Ansible Blocks and Error Handling

In playbooks, *blocks* are clauses that logically group tasks, and can be used to control how tasks are executed. For example, a task **block** can have a **when** directive to apply a conditional to multiple tasks:

```
- name: block example
hosts: all
tasks:
  - block:
    - name: package needed by yum
      yum:
        name: yum-plugin-versionlock
        state: present
    - name: lock version of tzdata
      lineinfile:
        dest: /etc/yum/pluginconf.d/versionlock.list
        line: tzdata-2016j-1
        state: present
  when: ansible_distribution == "RedHat"
```

Blocks also allow for error handling in combination with the **rescue** and **always** statements. If any task in a **block** fails, tasks in its **rescue** block are executed in order to recover. After the tasks in the block and possibly the rescue run, then tasks in its **always** block run. To summarize:

- **block**: Defines the main tasks to run.
- **rescue**: Defines the tasks that will be run if the tasks defined in the **block** clause fails.
- **always**: Defines the tasks that will always run independently of the success or failure of tasks defined in the **block** and **rescue** clauses.

The following example shows how to implement a block in a playbook. Even if tasks defined in the **block** clause fail, tasks defined in the **rescue** and **always** clauses will be executed.

```
tasks:
  - block:
    - name: upgrade the database
      shell:
        cmd: /usr/local/lib/upgrade-database
  rescue:
    - name: revert the database upgrade
      shell:
        cmd: /usr/local/lib/revert-database
  always:
    - name: always restart the database
      service:
        name: mariadb
        state: restarted
```

The **when** condition on a **block** also applies to its **rescue** and **always** sections if present.



References

Error Handling in Playbooks – Ansible Documentation

http://docs.ansible.com/ansible/playbooks_error_handling.html

Error Handling – Blocks – Ansible Documentation

http://docs.ansible.com/ansible/playbooks_blocks.html#error-handling

Guided Exercise: Handling Errors

In this exercise, you will handle errors in Ansible playbooks using various features

Outcomes

You should be able to:

- Ignore failed commands during the execution of playbooks.
- Force execution of handlers.
- Override what constitutes a failure in tasks.
- Override the **changed** state for tasks.
- Implement blocks/rescue/always in playbooks.

Before you begin

From **workstation**, run the lab setup script to confirm the environment is ready for the lab to begin. The script creates the working directory, **dev-failures**.

```
[student@workstation ~]$ lab task-control-failures setup
```

Steps

1. From **workstation.lab.example.com**, change to the **dev-failures** project directory.

```
[student@workstation ~]$ cd ~/dev-failures
[student@workstation dev-failures]$
```

2. The lab script created an Ansible configuration file as well as an inventory file that contains the server, **servera.lab.example.com**, in the group, **databases**. Review the file before proceeding.
3. Create the **playbook.yml** playbook, which contains a play with two tasks. The first task is written with a deliberate error that will cause it to fail.
 - 3.1. Open the playbook in a text editor. Define three variables: **web_package** with a value of **http**, **db_package** with a value of **mariadb-server** and **db_service** with a value of **mariadb**. The variables will be used to install the required packages and start the server.

The **http** value is an intentional error in the package name. The file should read as follows:

```
---
- hosts: databases
  vars:
    web_package: http
    db_package: mariadb-server
    db_service: mariadb
```

3.2. Define two tasks that use the **yum** module and the two variables, **web_package** and **db_package**. The tasks will install the required packages. The tasks should read as follows:

```
tasks:
  - name: Install {{ web_package }} package
    yum:
      name: "{{ web_package }}"
      state: latest

  - name: Install {{ db_package }} package
    yum:
      name: "{{ db_package }}"
      state: latest
```

4. Run the playbook and watch the output of the play.

```
[student@workstation dev-failures]$ ansible-playbook playbook.yml

PLAY [databases] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Install http package] ****
fatal: [servera.lab.example.com]: FAILED! => {"changed": false, "failed": true, "msg": "No package matching 'http' found available, installed or updated", "rc": 126, "results": ["No package matching 'http' found available, installed or updated"]}
to retry, use: --limit @/home/student/dev-failures/playbook.retry

PLAY RECAP ****
servera.lab.example.com    : ok=1    changed=0    unreachable=0    failed=1
```

The task failed because there is no existing package called **http**. Because the first task failed, the second task was not run.

5. Update the first task to ignore any errors by adding the **ignore_errors** keyword. The tasks should read as follows:

```
tasks:
  - name: Install {{ web_package }} package
    yum:
      name: "{{ web_package }}"
      state: latest
    ignore_errors: yes

  - name: Install {{ db_package }} package
    yum:
      name: "{{ db_package }}"
      state: latest
```

6. Run the playbook another time and watch the output of the play.

```
[student@workstation dev-failures]$ ansible-playbook playbook.yml
```

```

PLAY [databases] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Install http package] ****
fatal: [servera.lab.example.com]: FAILED! => {"changed": false, "failed": true,
  "msg": "No Package matching 'http' found available, installed or updated",
  "rc": 0, "results": []}
...ignoring

TASK [Install mariadb-server package] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com : ok=3    changed=1    unreachable=0    failed=0

```

Despite the fact that the first task failed, Ansible executed the second one.

7. In this step, we'll set up a **block** directive so you can experiment with how they work.
 - 7.1. Update the playbook by nesting the first task in a **block** clause. Remove the line that sets **ignore_errors: yes**. The block should read as follows:

```

- block:
  - name: Install {{ web_package }} package
    yum:
      name: "{{ web_package }}"
      state: latest

```

- 7.2. Nest the task that installs the *mariadb-server* package in a **rescue** clause. The task will be executed if the the task listed in **block** fails. The block should read as follows:

```

rescue:
  - name: Install {{ db_package }} package
    yum:
      name: "{{ db_package }}"
      state: latest

```

- 7.3. Finally, add an **always** clause that will start the database server upon installation using the **service** module. The clause should read as follows:

```

always:
  - name: Start {{ db_service }} service
    service:
      name: "{{ db_service }}"
      state: started

```

- 7.4. Once updated, the task section should read as follows:

```

tasks:
  - block:
    - name: Install {{ web_package }} package
      yum:
        name: "{{ web_package }}"
        state: latest
  rescue:

```

```
- name: Install {{ db_package }} package
  yum:
    name: "{{ db_package }}"
    state: latest
  always:
    - name: Start {{ db_service }} service
      service:
        name: "{{ db_service }}"
        state: started
```

8. Now we'll run the playbook again to see what happens.

8.1. Run the playbook. The task in the **block** that makes sure the **web_package** is installed will fail, which will cause the task in the **rescue** block to run. Then the task in the **always** block will run.

```
[student@workstation dev-failures]$ ansible-playbook playbook.yml

PLAY [databases] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Install http package] ****
fatal: [servera.lab.example.com]: FAILED! => {"changed": false, "failed": true,
  "msg": "No package matching 'http' found available, installed or updated",
  "rc": 126, "results": ["No package matching 'http' found available, installed
or updated"]}

TASK [Install mariadb-server package] ****
ok: [servera.lab.example.com]

TASK [Start mariadb service] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com : ok=3    changed=1    unreachable=0    failed=1
```

8.2. Edit the playbook, correcting the value of the **web_package** variable to read **httpd**. That will cause the task in the block to succeed the next time we run the playbook.

```
vars:
  web_package: httpd
  db_package: mariadb-server
  db_service: mariadb
```

8.3. Run the playbook again. This time, the task in the **block** will not fail. This will cause the task in the **rescue** to be ignored. The task in the **always** will still run.

```
[student@workstation dev-failures]$ ansible-playbook playbook.yml

PLAY [databases] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Install httpd package] ****
changed: [servera.lab.example.com]
```

```

TASK [Start mariadb service] ****
ok: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com : ok=3    changed=1    unreachable=0    failed=0

```

9. This step will explore how to control the condition that will cause a task to report it "changed" the managed host.
- 9.1. Edit the playbook to add two tasks to the start of the play, preceding the **block**. The first task will use the **command** module to run the **date** command and register the result in the **command_result** variable. The second task will use the **debug** module to print the standard output of the first task's command.

```

tasks:
  - name: Check local time
    command: date
    register: command_result

  - name: Print local time
    debug:
      var: command_result["stdout"]

```

- 9.2. Run the playbook. You should see that the first task which runs the **command** module reports "changed", even though it didn't change the remote system, it only collected information about the time. That is because the **command** module isn't "smart" enough to tell the difference between a command that collects data and a command that changes state.

```

[student@workstation dev-failures]$ ansible-playbook playbook.yml

PLAY [databases] ****

TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Check local time] ****
changed: [servera.lab.example.com]

TASK [Print local time] ****
ok: [servera.lab.example.com] => {
    "command_result["stdout"]": "Thu Jul 20 03:40:34 UTC 2017"
}

TASK [Install httpd package] ****
ok: [servera.lab.example.com]

TASK [Start mariadb service] ****
ok: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com : ok=5    changed=1    unreachable=0    failed=0

```

If you run the playbook again, the "Check local time" task will report "changed" every time.

- 9.3. That **command** task shouldn't report "changed" every time it runs because it's not changing the managed host. Since you know that the task will never change a managed host, add the line **changed_when: false** to the task to suppress the change.

```
tasks:
  - name: Check local time
    command: date
    register: command_result
    changed_when: false

  - name: Print local time
    debug:
      var: command_result["stdout"]
```

- 9.4. Run the playbook one more time to see that the task now reports "ok", but the task is still being run and is still saving the time in the variable.

```
[student@workstation dev-failures]$ ansible-playbook playbook.yml

PLAY [databases] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Check local time] ****
ok: [servera.lab.example.com]

TASK [Print local time] ****
ok: [servera.lab.example.com] => {
    "command_result[\"stdout\"]": "Thu Jul 20 03:42:12 UTC 2017"
}

TASK [Install httpd package] ****
ok: [servera.lab.example.com]

TASK [Start mariadb service] ****
ok: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com    : ok=5      changed=0      unreachable=0      failed=0
```

10. As a final exercise, edit the playbook to explore how the **failed_when** directive interacts with tasks.
- 10.1. Edit the "Install {{ web_package }} package" task so that it reports as having failed when **web_package** has the value **httpd**. Since this is the case, the task will report failure when we run the play.

Be careful with your indentation to make sure the directive is correctly set on the task.

```
- block:
  - name: Install {{ web_package }} package
    yum:
      name: "{{ web_package }}"
      state: latest
    failed_when: web_package == "httpd"
```

10.2.Run the playbook.

```
[student@workstation dev-failures]$ ansible-playbook playbook.yml

PLAY [databases] *****

TASK [Gathering Facts] *****
ok: [servera.lab.example.com]

TASK [Check local time] *****
ok: [servera.lab.example.com]

TASK [Print local time] *****
ok: [servera.lab.example.com] => {
    "command_result[\"stdout\"]": "Thu Jul 20 03:53:01 UTC 2017"
}

TASK [Install httpd package] *****
fatal: [servera.lab.example.com]: FAILED! => {"changed": false, "failed": true,
"failed_when_result": true, "msg": "", "rc": 0, "results": ["All packages
providing httpd are up to date", ""]}

TASK [Install mariadb-server package] *****
ok: [servera.lab.example.com]

TASK [Start mariadb service] *****
ok: [servera.lab.example.com]

PLAY RECAP *****
servera.lab.example.com      : ok=5      changed=0      unreachable=0      failed=1
```

Look carefully at the output. The "Install httpd package" task *reports* that it failed, but it actually ran and made sure the package is installed first! The **failed_when** directive changes the status the task reports *after* the task runs, it does not change the behavior of the task itself.

However, the failure reported might change the behavior of the rest of the play. Since that task was in a **block** and reported that it failed, the "Install mariadb-server package" task in the block's **rescue** section was run.

Cleanup

Run the **lab task-control-failures cleanup** command to cleanup after the lab.

```
[student@workstation ~]$ lab task-control-failures cleanup
```

Lab: Implementing Task Control

In this lab, you will install the Apache web server and secure it using `mod_ssl`. You will use various Ansible conditionals to deploy the environment.

Outcomes

You should be able to:

- Define conditionals in Ansible playbooks.
- Set up loops that iterate over elements.
- Define handlers in playbooks.
- Define tags and use them in conditionals.
- Handle errors in playbooks.

Before you begin

Log in as the `student` user on `workstation` and run `lab task-control setup`. This setup script ensures that the managed host, `serverb`, is reachable on the network. It also ensures that the correct Ansible configuration file and inventory are installed on the control node.

```
[student@workstation ~]$ lab task-control setup
```

Steps

1. From `workstation.lab.example.com`, change to the `lab-task-control` project directory.
2. **Defining tasks for the web server**

In the top-level directory for this lab, create the `install_packages.yml` task file. Define a task that installs the latest version of the `httpd` and `mod_ssl` packages. For the two packages to install, use the variables called `web_package` and `ssl_package`. The variables will be defined later on in the main playbook. Use a loop for installing the packages; moreover, the packages should only be installed if the server belongs to the `webservers` group, and only if the available memory on the system is greater than the amount of memory the `memory` variable defines. The variable will be set upon import of the task; use an Ansible fact to determine the available memory on the managed host.

Add a task that starts the service defined by the `web_service` variable. That variable will be set in the main playbook.

3. **Defining tasks for the web server's configuration**

Create the `configure_web.yml` task file. Add a task that checks whether or not the `httpd` package is installed and register the output in a variable. Update the condition to consider the task as failed based on the return code of the command (the return code is `1` when a package is not installed).

Create a block that executes only if the `httpd` package is installed (use the return code that has been captured in the first task). The block should start with a task that retrieves the file

that the **https_uri** variable defines (the variable will be set in the main playbook) and copy it to **serverb.lab.example.com** in the **/etc/httpd/conf.d/** directory.

Define a task that creates the **ssl** directory under **/etc/httpd/conf.d/** on the managed host with a mode of **0755**. The directory will store the SSL certificates.

Define a task that creates the **logs** directory under **/var/www/html/** on the managed host with a mode of **0755**. The directory will store the SSL logs.

Define a task that uses the **stat** module to ensure the **/etc/httpd/conf.d/ssl.conf** file exists, and capture the output in a variable. Define a task that renames the **/etc/httpd/conf.d/ssl.conf** file as **/etc/httpd/conf.d/ssl.conf.bak**, only if the file exists (use the captured output from the previous task).

Define another task that retrieves and extracts the SSL certificates file that the **ssl_uri** variable defines (the variable will be set in the main playbook). Extract the file under the **/etc/httpd/conf.d/ssl/** directory.

Configure this task to notify the **restart_services** handler.

Define the task that creates the **index.html** file under the **/var/www/html/** directory. The file should use Ansible facts and should read as follows:

```
serverb.lab.example.com (172.25.250.11) has been customized by Ansible
```

4. Defining tasks for the firewall

Create the **configure_firewall.yml** task file. Start with a task that installs the package that the **fw_package** variable defines (the variable will be set in the main playbook). Create a task that starts the service specified by the **fw_service** variable.

Create a task that adds firewall rules for the **http** and **https** services using a loop. The rules should be applied immediately and persistently. Tag all tasks with the **production** tag.

5. Defining the main playbook

In the top-level project directory for this lab, create the main playbook, **playbook.yml**. The playbook should only apply to hosts in the **webservers** group. Define a block that imports the following three task files: **install_packages.yml**, **configure_web.yml**, and **configure_firewall.yml**.

For the task that imports the **install_packages.yml** playbook, define the following variables: **memory** with a value of **256**, **web_package** with a value of **httpd**, **ssl_package** with a value of **mod_ssl** and **web_service** with a value of **httpd**.

For the task that imports the **configure_web.yml** file, define the following variables: **https_uri** with a value of **http://materials.example.com/task_control/https.conf**, and **ssl_uri** with a value of **http://materials.example.com/task_control/ssl.tar.gz**.

For the task that imports the **configure_firewall.yml** playbook, add a condition to only import the tasks tagged with the **production** tag. Define the **fw_package** and **fw_service** variables with a value of **firewalld**.

In the **rescue** clause for the block, define a task to install the **httpd** service. Notify the **restart_services** handler to start the service upon its installation.

Add a **debug** statement that reads:

```
Failed to import and run all the tasks; installing the web server manually
```

Add an **always** statement that uses the **shell** module to query the status of the **httpd** service using **systemctl**.

Define the **restart_services** handle to restart both the **httpd** and **firewalld** services using a loop.

6. Executing the `playbook.yml` playbook

Run the **playbook.yml** playbook to set up the environment. Ensure the web server has been correctly configured by querying the home page of the web server (**https://serverb.example.com**) using **curl** with the **-k** option to allow insecure connections. The output should read as follows:

```
serverb.lab.example.com (172.25.250.11) has been customized by Ansible
```

Evaluation

Run the **lab task-control grade** command from **workstation** to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab task-control grade
```

Cleanup

Run the **lab task-control cleanup** command to cleanup after the lab.

```
[student@workstation ~]$ lab task-control cleanup
```

Solution

In this lab, you will install the Apache web server and secure it using **mod_ssl**. You will use various Ansible conditionals to deploy the environment.

Outcomes

You should be able to:

- Define conditionals in Ansible playbooks.
- Set up loops that iterate over elements.
- Define handlers in playbooks.
- Define tags and use them in conditionals.
- Handle errors in playbooks.

Before you begin

Log in as the **student** user on **workstation** and run **lab task-control setup**. This setup script ensures that the managed host, **serverb**, is reachable on the network. It also ensures that the correct Ansible configuration file and inventory are installed on the control node.

```
[student@workstation ~]$ lab task-control setup
```

Steps

1. From **workstation.lab.example.com**, change to the **lab-task-control** project directory.

```
[student@workstation ~]$ cd ~/lab-task-control  
[student@workstation lab-task-control]$
```

2. **Defining tasks for the web server**

In the top-level directory for this lab, create the **install_packages.yml** task file. Define a task that installs the latest version of the *httpd* and *mod_ssl* packages. For the two packages to install, use the variables called **web_package** and **ssl_package**. The variables will be defined later on in the main playbook. Use a loop for installing the packages; moreover, the packages should only be installed if the server belongs to the **webservers** group, and only if the available memory on the system is greater than the amount of memory the **memory** variable defines. The variable will be set upon import of the task; use an Ansible fact to determine the available memory on the managed host.

Add a task that starts the service defined by the **web_service** variable. That variable will be set in the main playbook.

- 2.1. The following steps will edit the single **install_packages.yml** task file.

In the top-level directory for this lab, create the **install_packages.yml** task file. Start by defining the task that uses the **yum** module in order to install the required packages. For the packages, use a loop and two variables: **web_package** and **ssl_package**. The two variables will be set by the main playbook.

```
---
```

```
- name: Installs the required packages
  yum:
    name: "{{ item }}"
  with_items:
    - "{{ web_package }}"
    - "{{ ssl_package }}"
```

- 2.2. Continue editing **install_packages.yml**. Add a **when** clause in order to install the packages only if:

1. The managed host is in the **webservers** group.
2. The amount of memory on the managed host is greater than the amount the **memory** variable defines. For the amount of memory the system has, the **ansible_memory_mb.real.total** can be used.

The **when** clause should read as follows:

```
when:
  - inventory_hostname in groups["webservers"]
  - "(ansible_memory_mb.real.total) > (memory)"
```

- 2.3. Finally, add the task that starts the service defined by the **web_service** variable (the variable will be set in the main playbook).

```
- name: Starts the service
  service:
    name: "{{ web_service }}"
    state: started
```

- 2.4. When completed, the file should read as follows:

```
---
- name: Installs the required packages
  yum:
    name: "{{ item }}"
  with_items:
    - "{{ web_package }}"
    - "{{ ssl_package }}"
  when:
    - inventory_hostname in groups["webservers"]
    - "(ansible_memory_mb.real.total) > (memory)"

- name: Starts the service
  service:
    name: "{{ web_service }}"
    state: started
```

3. Defining tasks for the web server's configuration

Create the **configure_web.yml** task file. Add a task that checks whether or not the *httpd* package is installed and register the output in a variable. Update the condition to consider the task as failed based on the return code of the command (the return code is **1** when a package is not installed).

Create a block that executes only if the `httpd` package is installed (use the return code that has been captured in the first task). The block should start with a task that retrieves the file that the `https_uri` variable defines (the variable will be set in the main playbook) and copy it to `serverb.lab.example.com` in the `/etc/httpd/conf.d/` directory.

Define a task that creates the `ssl` directory under `/etc/httpd/conf.d/` on the managed host with a mode of **0755**. The directory will store the SSL certificates.

Define a task that creates the `logs` directory under `/var/www/html/` on the managed host with a mode of **0755**. The directory will store the SSL logs.

Define a task that uses the `stat` module to ensure the `/etc/httpd/conf.d/ssl.conf` file exists, and capture the output in a variable. Define a task that renames the `/etc/httpd/conf.d/ssl.conf` file as `/etc/httpd/conf.d/ssl.conf.bak`, only if the file exists (use the captured output from the previous task).

Define another task that retrieves and extracts the SSL certificates file that the `ssl_uri` variable defines (the variable will be set in the main playbook). Extract the file under the `/etc/httpd/conf.d/ssl/` directory.

Configure this task to notify the `restart_services` handler.

Define the task that creates the `index.html` file under the `/var/www/html/` directory. The file should use Ansible facts and should read as follows:

```
serverb.lab.example.com (172.25.250.11) has been customized by Ansible
```

3.1. The following steps will edit the single `configure_web.yml` file.

In the top-level directory of this lab, create the `configure_web.yml` tasks file. Start with a task that uses the `shell` module to determine whether or not the `httpd` package is installed. The `failed_when` variable will be used to override how Ansible should consider the task as failed by using the return code.

```
---
- shell:
    rpm -q httpd
    register: rpm_check
    failed_when: rpm_check.rc == 1
```

3.2. Continue editing the `configure_web.yml` file. Create a block that contains the tasks for configuring the files. Start the block with a task that uses the `get_url` module to retrieve the Apache SSL configuration file. Use the `https_uri` variable for the `url` and `/etc/httpd/conf.d/` for the remote path on the managed host.

```
- block:
  - get_url:
      url: "{{ https_uri }}"
      dest: /etc/httpd/conf.d/
```

3.3. Create the `/etc/httpd/conf.d/ssl` remote directory with a mode of **0755**.

```
- file:  
  path: /etc/httpd/conf.d/ssl  
  state: directory  
  mode: 0755
```

- 3.4. Create the **/var/www/html/logs** remote directory with a mode of **0755**.

```
- file:  
  path: /var/www/html/logs  
  state: directory  
  mode: 0755
```

- 3.5. Ensure the **/etc/httpd/conf.d/ssl.conf** file exists. Capture the output in the **ssl_file** variable using the **register** statement.

```
- stat:  
  path: /etc/httpd/conf.d/ssl.conf  
  register: ssl_file
```

- 3.6. Create the task that renames the **/etc/httpd/conf.d/ssl.conf** file as **/etc/httpd/conf.d/ssl.conf.bak**. The task will evaluate the content of the **ssl_file** variable before attempting to rename the file.

```
- shell:  
  mv /etc/httpd/conf.d/ssl.conf /etc/httpd/conf.d/ssl.conf.bak  
  when: ssl_file.stat.exists
```

- 3.7. Create the task that uses the **unarchive** module to retrieve the remote SSL configuration files. Use the **ssl_uri** variable for the source and **/etc/httpd/conf.d/ssl/** as the destination. Instruct the task to notify the **restart_services** handler when the file has been copied.

```
- unarchive:  
  src: "{{ ssl_uri }}"  
  dest: /etc/httpd/conf.d/ssl/  
  copy: no  
  notify:  
    - restart_services
```

- 3.8. Add the last task that creates the **index.html** file under **/var/www/html/** on the managed host. The page should read as follows:

```
severb.lab.example.com (172.25.250.11) has been customized by Ansible
```

Use the two following Ansible facts to create the page: **ansible_fqdn**, and **ansible_default_ipv4.address**.

```
- copy:  
  content: "{{ ansible_fqdn }} ({{ ansible_default_ipv4.address }}) has been  
  customized by Ansible\n"
```

```
dest: /var/www/html/index.html
```

- 3.9. Finally, make sure the block only runs if the *httpd* package is installed. To do so, add a **when** clause that parses the return code contained in the **rpm_check** registered variable.

```
when:
  rpm_check.rc == 0
```

- 3.10. When completed, the file should read as follows:

```
---
- shell:
    rpm -q httpd
  register: rpm_check
  failed_when: rpm_check.rc == 1

- block:
  - get_url:
      url: "{{ https_uri }}"
      dest: /etc/httpd/conf.d/

  - file:
      path: /etc/httpd/conf.d/ssl
      state: directory
      mode: 0755

  - file:
      path: /var/www/html/logs
      state: directory
      mode: 0755

  - stat:
      path: /etc/httpd/conf.d/ssl.conf
  register: ssl_file

  - shell:
      mv /etc/httpd/conf.d/ssl.conf /etc/httpd/conf.d/ssl.conf.bak
  when: ssl_file.stat.exists

  - unarchive:
      src: "{{ ssl_uri }}"
      dest: /etc/httpd/conf.d/ssl/
      copy: no
  notify:
    - restart_services

  - copy:
      content: "{{ ansible_fqdn }} ({{ ansible_default_ipv4.address }}) has been
customized by Ansible\n"
      dest: /var/www/html/index.html

when:
  rpm_check.rc == 0
```

4. Defining tasks for the firewall

Create the **configure_firewall.yml** task file. Start with a task that installs the package that the **fw_package** variable defines (the variable will be set in the main playbook). Create a task that starts the service specified by the **fw_service** variable.

Create a task that adds firewall rules for the **http** and **https** services using a loop. The rules should be applied immediately and persistently. Tag all tasks with the **production** tag.

4.1. The following steps will edit the single **configure_firewall.yml** file.

Define the task that uses the **yum** module to install latest version of the firewall service. Tag the task with the **production** tag. The task should read as follows:

```
---
```

- yum:
 - name: "{{ fw_package }}"
 - state: latest
 - tags: production

4.2. Continue editing the **configure_firewall.yml** file. Add the task that starts the firewall service using the **fw_service** variable and tag it as **production**. The task should read as follows:

```
- service:

- name: "{{ fw_service }}"
- state: started
- tags: production

```

4.3. Write the task that uses the **firewalld** module to add the **http** and **https** service rules to the firewall. The rules should be applied immediately as well as persistently. Use a loop for the two rules. Tag the task as **production**.

```
- firewalld:

- service: "{{ item }}"
- immediate: true
- permanent: true
- state: enabled


- with_items:
  - http
  - https

- tags: production

```

4.4. When completed, the file should read as follows:

```
---
```

- yum:
 - name: "{{ fw_package }}"
 - state: latest
 - tags: production
- service:
 - name: "{{ fw_service }}"
 - state: started
 - tags: production
- firewalld:

```

service: "{{ item }}"
immediate: true
permanent: true
state: enabled
with_items:
  - http
  - https
tags: production

```

5. Defining the main playbook

In the top-level project directory for this lab, create the main playbook, **playbook.yml**. The playbook should only apply to hosts in the **webservers** group. Define a block that imports the following three task files: **install_packages.yml**, **configure_web.yml**, and **configure_firewall.yml**.

For the task that imports the **install_packages.yml** playbook, define the following variables: **memory** with a value of **256**, **web_package** with a value of **httpd**, **ssl_package** with a value of **mod_ssl** and **web_service** with a value of **httpd**.

For the task that imports the **configure_web.yml** file, define the following variables: **https_uri** with a value of **http://materials.example.com/task_control/**, **https.conf**, and **ssl_uri** with a value of **http://materials.example.com/task_control/ssl.tar.gz**.

For the task that imports the **configure_firewall.yml** playbook, add a condition to only import the tasks tagged with the **production** tag. Define the **fw_package** and **fw_service** variables with a value of **firewalld**.

In the **rescue** clause for the block, define a task to install the **httpd** service. Notify the **restart_services** handler to start the service upon its installation.

Add a **debug** statement that reads:

```
Failed to import and run all the tasks; installing the web server manually
```

Add an **always** statement that uses the **shell** module to query the status of the **httpd** service using **systemctl**.

Define the **restart_services** handle to restart both the **httpd** and **firewalld** services using a loop.

5.1. The following steps will edit the single **playbook.yml** file.

Create the **playbook.yml** playbook and start by targeting the hosts in the **webservers** host group.

```

---
- hosts: webservers

```

5.2. Continue editing the **playbook.yml** file. Create a block for importing the three task files, using the **include** statement. For the first **include**, use **install_packages.yml** as the name of the file to import. Define the four variables required by the file:

1. **memory**, with a value of **256**
2. **web_package**, with a value of **httpd**
3. **ssl_package**, with a value of **mod_ssl**
4. **web_service**, with a value of **httpd**

Add the following to the **playbook.yml** file:

```
tasks:  
  - block:  
    - include: install_packages.yml  
    vars:  
      memory: 256  
      web_package: httpd  
      ssl_package: mod_ssl  
      web_service: httpd
```

- 5.3. For the second **include**, use **configure_web.yml** as the name of the file to import.

Define the two variables required by the file:

- **https_uri**, with a value of **http://materials.example.com/task_control/https.conf**
- **ssl_uri**, with a value of **http://materials.example.com/task_control/ssl.tar.gz**

Add the following to the **playbook.yml** file:

```
  - include: configure_web.yml  
  vars:  
    https_uri: http://materials.example.com/task_control/https.conf  
    ssl_uri: http://materials.example.com/task_control/ssl.tar.gz
```

- 5.4. For the third **include**, use **configure_firewall.yml** as the name of the file to import. Define the variables required by the file, **fw_package** and **fw_service**, both with a value of **firewalld**. Import only the tasks that are tagged with **production**.

```
  - include: configure_firewall.yml  
  vars:  
    fw_package: firewalld  
    fw_service: firewalld  
  tags: production
```

- 5.5. Create the **rescue** clause for the block that installs the latest version of the **httpd** package and notifies the **restart_services** handler upon the package installation. Add a **debug** statement that reads:

```
Failed to import and run all the tasks; installing the web server manually
```

```
rescue:  
  - yum:  
    name: httpd  
    state: latest
```

```

    notify:
      - restart_services

    - debug:
        msg: "Failed to import and run all the tasks; installing the web
server manually"

```

- 5.6. In the **always** clause, use the **shell** module to query the status of the **httpd** service, using **systemctl**. Add the following to the **playbook.yml** file:

```

always:
  - shell:
      cmd: "systemctl status httpd"

```

- 5.7. Define the **restart_services** handler that uses a loop to restart both the **firewalld** and **httpd** services. Add the following to the **playbook.yml** file:

```

handlers:
  - name: restart_services
    service:
      name: "{{ item }}"
      state: restarted
    with_items:
      - httpd
      - firewalld

```

- 5.8. When completed, the playbook should read as follows:

```

---
- hosts: webservers
  tasks:
    - block:
        - include: install_packages.yml
          vars:
            memory: 256
            web_package: httpd
            ssl_package: mod_ssl
            web_service: httpd
        - include: configure_web.yml
          vars:
            https_uri: http://materials.example.com/task_control/https.conf
            ssl_uri: http://materials.example.com/task_control/ssl.tar.gz
        - include: configure_firewall.yml
          vars:
            fw_package: firewalld
            fw_service: firewalld
          tags: production

    rescue:
      - yum:
          name: httpd
          state: latest
        notify:
          - restart_services

    - debug:
        msg: "Failed to import and run all the tasks; installing the web
server manually"

```

```
always:
  - shell:
    cmd: "systemctl status httpd"

handlers:
  - name: restart_services
    service:
      name: "{{ item }}"
      state: restarted
    with_items:
      - httpd
      - firewalld
```

6. Executing the `playbook.yml` playbook

Run the `playbook.yml` playbook to set up the environment. Ensure the web server has been correctly configured by querying the home page of the web server (`https://serverb.example.com`) using `curl` with the `-k` option to allow insecure connections. The output should read as follows:

```
serverb.lab.example.com (172.25.250.11) has been customized by Ansible
```

6.1. Before running the playbook, verify its syntax is correct by running `ansible-playbook --syntax-check`. If it reports any errors, correct them before moving to the next step. You should see output similar to the following:

```
[student@workstation lab-task-control]$ ansible-playbook --syntax-check
playbook.yml

playbook: playbook.yml
```

6.2. Using the `ansible-playbook` command, run the `playbook.yml` playbook. The playbook should:

- Import and run the tasks that install the web server packages only if there is enough memory on the managed host.
- Import and run the tasks that configure SSL for the web server.
- Import and run the tasks that create the firewall rule for the web server to be reachable.

```
[student@workstation lab-task-control]$ ansible-playbook playbook.yml
PLAY [webservers] ****
TASK [Gathering Facts] ****
ok: [serverb.lab.example.com]
...
RUNNING HANDLER [restart_services] ****
changed: [serverb.lab.example.com] => (item=httpd)
changed: [serverb.lab.example.com] => (item=firewalld)

PLAY RECAP ****
serverb.lab.example.com : ok=16    changed=14    unreachable=0    failed=0
```

- 6.3. Use **curl** to confirm the web page is available. The **-k** option allows you to bypass any SSL strict checking.

```
[student@workstation lab-task-control]$ curl -k https://serverb.lab.example.com  
serverb.lab.example.com (172.25.250.11) has been customized by Ansible
```

Evaluation

Run the **lab task-control grade** command from **workstation** to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab task-control grade
```

Cleanup

Run the **lab task-control cleanup** command to cleanup after the lab.

```
[student@workstation ~]$ lab task-control cleanup
```

Summary

In this chapter, you learned:

- *Loops* can be used to iterate over a set of values. They can be a list of items in sequence or random order, files, an autogenerated sequence of numbers, or other things
- *Conditionals* can be used to execute tasks or plays only when certain conditions have been met
- Conditions can be tested with various operators, including string comparisons, mathematical operators, and Boolean values
- *Handlers* are special tasks that execute at the end of the play if notified by other tasks
- *Tags* are used to mark certain tasks to be skipped or executed based on what tags a task has
- Tasks can be configured to handle error conditions by ignoring task failure, forcing handlers to be called even if the task failed, mark a task as failed when it succeeded, or override the behavior that causes a task to be marked as changed
- *Blocks* can be used to group tasks as a unit and execute other tasks depending on whether or not all the tasks in the block succeed



CHAPTER 6

IMPLEMENTING JINJA2 TEMPLATES

Overview	
Goal	Implement a Jinja2 template
Objectives	<ul style="list-style-type: none">• Describe Jinja2 templates• Implement Jinja2 templates
Sections	<ul style="list-style-type: none">• Describing Jinja2 Templates (and Quiz)• Implementing Jinja2 Templates (and Guided Exercise)
Lab	<ul style="list-style-type: none">• Implementing Jinja2 Templates

Describing Jinja2 Templates

Objectives

After completing this section, students should be able to:

- Describe Jinja2 templates
- Describe the differences between YAML files and Jinja2 templates
- Describe variables, control structures, and comment use in Jinja2 templates

Introduction to Jinja2

Ansible uses the Jinja2 templating system to modify files before they are distributed to managed hosts. Generally speaking, it is preferable to avoid modifying configuration files through logic in templates. However, templates can be useful when systems need to have slightly modified versions of the same file. Ansible also uses Jinja2 to reference variables in playbooks.

Ansible allows Jinja2 loops and conditionals to be used in templates, but they are not allowed in playbooks. Ansible playbooks are completely machine-parseable YAML. This is an important feature, because it means it is possible to have code generate pieces of files, or to have other third-party tools read Ansible files. Not everyone will need this feature, but it can unlock interesting possibilities.

Delimiters

Variables or logic expressions are placed between tags, or delimiters. For example, Jinja2 templates use `{% EXPRESSION %}` for expressions or logic (for example, loops), while `{{ EXPRESSION }}` are used for outputting the results of an expression or a variable to the end user. The latter tag, when rendered, is replaced with a value or values, and are seen by the end user. Use `{# COMMENT #}` syntax to enclose comments.

In the following example the first line includes a comment that will not be included in the final file. The variable references in the second line are replaced with the values of the system facts being referenced.

```
{# /etc/hosts line #
{{ ansible_default_ipv4.address }}    {{ ansible_hostname }}
```

Control Structures

Often the result of a play depends on the value of a variable, fact (something learned about the remote system), or previous task result. In some cases, the values of variables depend on other variables. Further, additional groups can be created to managed hosts based on whether the hosts match other criteria. There are many options to control execution flow in Ansible.

Loops

Jinja2 uses the `for` statement to provide looping functionality. In the following example, the `user` variable is replaced with all the values included in `users`.

```
{% for user in users %}
{{ user }}
```

```
{% endfor %}
```

The following **for** statement runs through all the values in the **users** variable, replacing **myuser** with each value, except when the value is **Snoopy**.

```
{# for statement #}
{% for myuser in users if not myuser == "Snoopy"%}
{{loop.index}} - {{ myuser }}
{% endfor %}
```

The **loop.index** variable expands to the index number that the loop is currently on. It has a value of 1 the first time the loop executes, and it increments by 1 through each iteration.

Conditionals

Jinja2 uses the **if** statement to provide conditional control. In the following example, the result is displayed if the **finished** variable is **True**.

```
{% if finished %}
  {{ result }}
{% endif %}
```

Variable Filters

Jinja2 provides filters which change the output format for template expressions (for example, to JSON). There are filters available for languages such as YAML or JSON. The **to_json** filter formats the expression output using JSON, and the **to_yaml** filter uses YAML as the formatting syntax.

```
{{ output | to_json }}
{{ output | to_yaml }}
```

Additional filters are available, such as the **to_nice_json** and **to_nice_yaml** filters, which format the expression output in either JSON or YAML human readable format.

```
{{ output | to_nice_json }}
{{ output | to_nice_yaml }}
```

Both the **from_json** and **from_yaml** filters expect a string in either JSON or YAML format and parse it.

```
{{ output | from_json }}
{{ output | from_yaml }}
```

The expressions used with **when** clauses in Ansible playbooks are Jinja2 expressions. Built-in Ansible filters that are used to test return values include **failed**, **changed**, **succeeded**, and **skipped**. The following task shows how filters can be used inside of conditional expressions.

```
tasks:
... Output omitted ...
- debug: msg="the execution was aborted"
  when: returnvalue | failed
```

Issues to be Aware of with YAML vs. Jinja2 in Ansible

The use of some Jinja2 expressions inside of a YAML playbook may change the meaning for those expressions, so they require some adjustments in the syntax used.

1. YAML syntax requires quotes when a value starts with a variable reference (`{{ }}`). The quotes prevent the parser from treating the expression as the start of a YAML dictionary. For example, the following playbook snippet will fail:

```
- hosts: app_servers
  vars:
    app_path: {{ base_path }}/bin
```

Instead, use the following syntax:

```
- hosts: app_servers
  vars:
    app_path: "{{ base_path }}/bin"
```

2. When there is a need to include nested `{{...}}` elements, the braces around the inner ones must be removed. Consider the following playbook snippet:

```
- name: display the host value
  debug:
    msg: hostname = {{ params[ {{ host_ip }} ] }}, IPAddr = {{ host_ip }}
```

Ansible raises the following error when it tries to run it:

```
... Output omitted ...
TASK [display the host value] ****
fatal: [localhost]: FAILED! => {"failed": true, "msg": "template error
while templating string: expected token ':', got '}'. String: hostname =
{{ params[ {{ host_ip }} ] }}, IPAddr = {{ host_ip }}"}
... Output omitted ...
```

Use the following syntax instead. It will run without error.

```
- name: display the host value
  debug:
    msg: hostname = {{ params[ host_ip ] }}, IPAddr = {{ host_ip }}
```

Now the playbook will run without error.

References

Template Designer Documentation – Jinja2 Documentation
<http://jinja.pocoo.org/docs/dev/templates/>

Quiz: Describing Jinja2 Templates

Choose the correct answers to the following questions:

1. What is the main purpose of Jinja2 templates usage within Ansible?
 - a. Modify files before they are distributed to managed hosts
 - b. Provide an enhanced alternative to YAML for playbooks
 - c. Add conditional and loop use inside of playbooks
 - d. Support object use in templates

2. Which three features are included in the Jinja2 templates? (Choose three.)
 - a. Variable filters
 - b. Objects
 - c. Loops
 - d. Conditionals
 - e. Iterators

3. Which two delimiters are allowed in Jinja2 templates? (Choose two.)
 - a. {{ ... }} for variables
 - b. \${...\$} for expressions
 - c. {%- %} for expressions
 - d. {@ ... @} for variables
 - e. {{(...)}} for functions

4. Which three of the following filters are supported in Jinja2 templates? (Choose three.)
 - a. to_nice_json
 - b. from_nice_json
 - c. to_nice_yaml
 - d. from_nice_yaml
 - e. to_yaml

5. Which two lines show valid Jinja2 usage in YAML? (Choose two.)
 - a. remote_host: "{{ host_name }}"
 - b. remote_host: {{ "host_name" }}
 - c. remote_host: params[{{ host_name }}]
 - d. remote_host: "{{ params[host_name] }}"

Solution

Choose the correct answers to the following questions:

1. What is the main purpose of Jinja2 templates usage within Ansible?
 - a. **Modify files before they are distributed to managed hosts**
 - b. Provide an enhanced alternative to YAML for playbooks
 - c. Add conditional and loop use inside of playbooks
 - d. Support object use in templates
2. Which three features are included in the Jinja2 templates? (Choose three.)
 - a. **Variable filters**
 - b. Objects
 - c. **Loops**
 - d. **Conditionals**
 - e. Iterators
3. Which two delimiters are allowed in Jinja2 templates? (Choose two.)
 - a. **{{ ... }} for variables**
 - b. **{\$... \$} for expressions**
 - c. **{% ... %} for expressions**
 - d. {@ ... @} for variables
 - e. {(...)} for functions
4. Which three of the following filters are supported in Jinja2 templates? (Choose three.)
 - a. **to_nice_json**
 - b. from_nice_json
 - c. **to_nice_yaml**
 - d. from_nice_yaml
 - e. **to_yaml**
5. Which two lines show valid Jinja2 usage in YAML? (Choose two.)
 - a. **remote_host: "{{ host_name }}"**
 - b. remote_host: {{ "host_name" }}
 - c. remote_host: params[{{ host_name }}]
 - d. **remote_host: "{{ params[host_name] }}"**

Implementing Jinja2 Templates

Objectives

After completing this section, students should be able to:

- Build a template file
- Use the template file in a playbook

Building a Jinja2 template

A Jinja2 template is composed of multiple elements: data, variables and expressions. Those variables and expressions are replaced with their values when the Jinja2 template is rendered. The variables used in the template can be specified in the **vars** section of the playbook. It is possible to use the managed hosts' facts as variables on a template.



Note

Remember that the facts associated with a managed host can be obtained using the **ansible system_hostname -i inventory_file -m setup** command.

The following example shows how to create a template with variables using two of the facts retrieved by Ansible from managed hosts: **ansible_hostname** and **ansible_date_time.date**. When the associated playbook is executed, those two facts will be replaced by their values in the managed host being configured.



Note

A file containing a Jinja2 template does not need any specific file extension (for example, **.j2**).

```
Welcome to {{ ansible_hostname }}.
Today's date is: {{ ansible_date_time.date }}.
```

The following example uses the **for** statement, and assumes a **myhosts** variable has been defined in the inventory file being used. This variable would contain a list of hosts to be managed. With the following **for** statement, all hosts in the **myhosts** group from the inventory would be listed.

```
{% for myhost in groups['myhosts'] %}
{{ myhost }}
{% endfor %}
```

It is recommended practice to include at the beginning of a Jinja2 template the **ansible_managed** variable, in order to reflect that the file is managed by Ansible. This will be reflected into the file created in the managed host, with a string including the date, user ID and managed host hostname. This variable by default has the following value.

```
ansible_managed = Ansible managed: {file} modified on %Y-%m-%d %H:%M:%S by {uid} on {host}
```

To include the **ansible_managed** variable inside a Jinja2 template, use the following syntax:

```
{{ ansible_managed }}
```

Using Jinja2 Templates in Playbooks

Jinja2 templates are a powerful tool to customize configuration files to be deployed on the managed hosts. When the Jinja2 template for a configuration file has been created, it can be deployed to the managed hosts using the **template** module, which supports the transfer of a local file in the control node to the managed hosts.

To use the **template** module, use the following syntax. The value associated with the **src** key specifies the source Jinja2 template, and the value associated to the **dest** key specifies the file to be created on the destination hosts.

```
tasks:  
  - name: template render  
    template:  
      src: /tmp/j2-template.j2  
      dest: /tmp/dest-config-file.txt
```



Note

The template module allows you to configure, in the destination file, settings such as the owner, group or mode, or validate it against a command (for example, **visudo -c**)

As seen in previous chapters, thanks to Jinja2 syntax, it is also possible to use variables, with or without filters, inside the YAML definition of a playbook, including facts.



References

template - Templates a file out to a remote server – Ansible Documentation
http://docs.ansible.com/ansible/template_module.html

Variables – Ansible Documentation
http://docs.ansible.com/ansible/playbooks_variables.html

Guided Exercise: Implementing Jinja2 Templates

In this exercise, you will create a simple template file that delivers a custom motd file.

Outcomes

You should be able to:

- Build a template file.
- Use the template file in a playbook.

Before you begin

Log in to **workstation** as **student** using **student** as a password.

On **workstation**, run the **lab_jinja2-implement setup** script. It checks if Ansible is installed on **workstation**, and creates the **/home/student/jinja2** directory and downloads the **ansible.cfg** file into it. It also downloads the **motd.yml**, **motd.j2**, and **inventory** files into the **/home/student/jinja2/files** directory.

```
[student@workstation ~]$ lab_jinja2-implement setup
```



Note

All the files used along this exercise are available on **workstation** in the **/home/student/jinja2/files** directory.

Steps

1. On workstation, go to the **/home/student/jinja2** directory.

```
[student@workstation ~]$ cd ~/jinja2/
```

2. Create the **inventory** file in the current directory. This file configures two groups: **webservers** and **workstations**. Include the system *servera.lab.example.com* in the **webservers** group, and the system *workstation.lab.example.com* in the **workstations** group.

```
[webservers]
servera.lab.example.com
```

```
[workstations]
workstation.lab.example.com
```

3. Create a template for the Message of the Day. Include it in the **motd.j2** file in the current directory. Include the following variables in the template:
 - **ansible_hostname** to retrieve the managed host hostname.
 - **ansible_date_time.date** for the managed host date.

- **system_owner** for the email of the owner of the system. This variable needs to be defined with an appropriate value in the **vars** section of the playbook template.

```
This is the system {{ ansible_hostname }}.  
Today's date is: {{ ansible_date_time.date }}.  
Only use this system with permission.  
You can ask {{ system_owner }} for access.
```

4. Create a playbook in a new file in the current directory, named **motd.yml**. Define the **system_owner** variable in the **vars** section, and include a task for the *template* module, which maps the **motd.j2** Jinja2 template to the remote file **/etc/motd** on the managed hosts. Set the owner and group to **root**, and the mode to **0644**.

```
---  
- hosts: all  
  user: devops  
  become: true  
  vars:  
    system_owner: clyde@example.com  
  tasks:  
    - template:  
        src: motd.j2  
        dest: /etc/motd  
        owner: root  
        group: root  
        mode: 0644
```

5. Before running the playbook, verify its syntax is correct by running **ansible-playbook --syntax-check**. If it reports any errors, correct them before moving to the next step. You should see output similar to the following:

```
[student@workstation jinja2]$ ansible-playbook --syntax-check motd.yml  
playbook: motd.yml
```

6. Run the playbook included in the **motd.yml** file.

```
[student@workstation jinja2]$ ansible-playbook motd.yml  
PLAY [all] *****  
TASK [Gathering Facts] *****  
ok: [servera.lab.example.com]  
ok: [workstation.lab.example.com]  
  
TASK [template] *****  
changed: [servera.lab.example.com]  
changed: [workstation.lab.example.com]  
  
PLAY RECAP *****  
servera.lab.example.com : ok=2    changed=1    unreachable=0    failed=0  
workstation.lab.example.com : ok=2    changed=1    unreachable=0    failed=0
```

7. Log in to **servera.lab.example.com** using the *devops* user, to verify the motd is displayed when logging in. Log out when you have finished.

```
[student@workstation jinja2]$ ssh devops@servera.lab.example.com
This is the system servera.
Today's date is: 2017-07-21.
Only use this system with permission.
You can ask clyde@example.com for access.
[devops@servera ~]# exit
Connection to servera.lab.example.com closed.
```

Evaluation

From **workstation**, run the **lab jinja2-implement grade** script to confirm success on this exercise.

```
[student@workstation ~]$ lab jinja2-implement grade
```

Cleanup

Run the **lab jinja2-implement cleanup** command to clean up after the lab.

```
[student@workstation jinja2]$ lab jinja2-implement cleanup
```

Lab: Implementing Jinja2 Templates

In this lab, you will create a file using a Jinja2 template in a playbook.

Outcomes

You should be able to:

- Build a template file.
- Use the template file in a playbook.

Before you begin

Log in to **workstation** as **student** using **student** as a password.

On **workstation**, run the **lab_jinja2-lab setup** script. It checks if Ansible is installed on **workstation**, and creates the **/home/student/jinja2-lab** directory and downloads the **ansible.cfg** file into it. It also downloads the **motd.yml**, **motd.j2**, and **inventory** files into the **/home/student/jinja2-lab/files** directory.

```
[student@workstation ~]$ lab_jinja2-lab setup
```



Note

All the files used in this exercise are available on **workstation** in the **/home/student/jinja2-lab/files** directory.

Steps

1. Create an inventory file, named **inventory**, in the **/home/student/jinja2-lab** directory. This inventory file defines the group **servers** which has the **serverb.lab.example.com** managed host associated to it.
2. Identify the facts on **serverb.lab.example.com** which show the status of the system memory.
3. Create a template for the Message of the Day, named **motd.j2**, in the current directory. Use the facts previously identified.
4. Create a new playbook file in the current directory, named **motd.yml**. Using the *template* module, configure the **motd.j2** Jinja2 template file previously created to map to the file **/etc/motd** on the managed hosts. This file has the **root** user as owner and group, and its permissions are **0644**. Configure the playbook so it uses the **devops** user, and sets the **become** parameter to be **true**.
5. Run the playbook included in the **motd.yml** file.
6. Check that the playbook included in the **motd.yml** file has been executed correctly.

Evaluation

From **workstation**, run the **lab_jinja2-lab grade** script to confirm success on this exercise.

```
[student@workstation ~]$ lab jinja2-lab grade
```

Cleanup

From **workstation**, run the **lab jinja2-lab cleanup** script to clean up after the lab.

```
[student@workstation ~]$ lab jinja2-lab cleanup
```

Solution

In this lab, you will create a file using a Jinja2 template in a playbook.

Outcomes

You should be able to:

- Build a template file.
- Use the template file in a playbook.

Before you begin

Log in to **workstation** as **student** using **student** as a password.

On **workstation**, run the **lab_jinja2-lab setup** script. It checks if Ansible is installed on **workstation**, and creates the **/home/student/jinja2-lab** directory and downloads the **ansible.cfg** file into it. It also downloads the **motd.yml**, **motd.j2**, and **inventory** files into the **/home/student/jinja2-lab/files** directory.

```
[student@workstation ~]$ lab_jinja2-lab setup
```



Note

All the files used in this exercise are available on **workstation** in the **/home/student/jinja2-lab/files** directory.

Steps

1. Create an inventory file, named **inventory**, in the **/home/student/jinja2-lab** directory. This inventory file defines the group **servers** which has the **serverb.lab.example.com** managed host associated to it.
 - 1.1. On workstation, go to the **/home/student/jinja2-lab** directory.

```
[student@workstation ~]$ cd ~/jinja2-lab/
```

 - 1.2. Create the **inventory** file in the current directory. This file configures one group: **servers**. Include the system **serverb.lab.example.com** in the **servers** group.

```
[servers]
serverb.lab.example.com
```

- 2. Identify the facts on **serverb.lab.example.com** which show the status of the system memory.
 - 2.1. Use the **setup** module to get a list of all the facts for the **serverb.lab.example.com** managed host. Both the **ansible_memfree_mb** and **ansible_memtotal_mb** facts provide information about the free memory and the total memory of the managed host.

```
[student@workstation jinja2-lab]$ ansible serverb.lab.example.com -m setup
serverb.lab.example.com | SUCCESS => {
    "ansible_facts": {
        ... Output omitted ...
    }
}
```

```

"ansible_memfree_mb": 157,
...
"ansible_memtotal_mb": 488,
...
},
"changed": false
}

```

3. Create a template for the Message of the Day, named **motd.j2**, in the current directory. Use the facts previously identified.

- 3.1. Create a new file, named **motd.j2**, in the current directory. Use both the **ansible_memfree_mb** and **ansible_memtotal_mb** fact variables to create a Message of the Day.

```

[student@workstation jinja2-lab]$ cat motd.j2
This system's total memory is: {{ ansible_memtotal_mb }} MBs.
The current free memory is: {{ ansible_memfree_mb }} MBs.

```

4. Create a new playbook file in the current directory, named **motd.yml**. Using the *template* module, configure the **motd.j2** Jinja2 template file previously created to map to the file **/etc/motd** on the managed hosts. This file has the *root* user as owner and group, and its permissions are **0644**. Configure the playbook so it uses the *devops* user, and sets the **become** parameter to be *true*.

- 4.1. Create a new playbook file in the current directory, named **motd.yml**. Using the *template* module, configure the **motd.j2** Jinja2 template file previously as the value for the **src** parameter, and **/etc/motd** as the value for the **dest** parameter. Configure the **owner** and **group** parameters to be *root*, and the **mode** parameter to be **0644**. Use the *devops* user for the **user** parameter, and configure the **become** parameter to be *true*.

```

[student@workstation jinja2-lab]$ cat motd.yml
---
- hosts: all
  user: devops
  become: true
  tasks:
    - template:
        src: motd.j2
        dest: /etc/motd
        owner: root
        group: root
        mode: 0644

```

5. Run the playbook included in the **motd.yml** file.

- 5.1. Before running the playbook, verify its syntax is correct by running **ansible-playbook --syntax-check**. If it reports any errors, correct them before moving to the next step. You should see output similar to the following:

```

[student@workstation jinja2-lab]$ ansible-playbook --syntax-check motd.yml
playbook: motd.yml

```

5.2. Run the playbook included in the **motd.yml** file.

```
[student@workstation jinja2-lab]$ ansible-playbook motd.yml
PLAY [all] ****
TASK [Gathering Facts] ****
ok: [serverb.lab.example.com]

TASK [template] ****
changed: [serverb.lab.example.com]

PLAY RECAP ****
serverb.lab.example.com : ok=2    changed=1    unreachable=0    failed=0
```

6. Check that the playbook included in the *motd.yml* file has been executed correctly.

6.1. Log in to **serverb.lab.example.com** using the *devops* user, to verify the motd is displayed when logging in. Log out when you have finished.

```
[student@workstation jinja2-lab]$ ssh devops@serverb.lab.example.com
This system's total memory is: 488 MBs.
The current free memory is: 162 MBs.
[devops@serverb ~]$ logout
```

Evaluation

From **workstation**, run the **lab jinja2-lab grade** script to confirm success on this exercise.

```
[student@workstation ~]$ lab jinja2-lab grade
```

Cleanup

From **workstation**, run the **lab jinja2-lab cleanup** script to clean up after the lab.

```
[student@workstation ~]$ lab jinja2-lab cleanup
```

Summary

In this chapter, you learned:

- Ansible uses the Jinja2 templating system to render files before they are distributed to managed hosts.
- YAML allows the use of Jinja2 based variables, with or without filters, inside the definition of a playbook.
- A Jinja2 template is usually composed of two elements: variables and expressions. Those variables and expressions are replaced with their values when the Jinja2 template is rendered.
- Filters in Jinja2 are a way of transforming template expressions from one kind of data into another.



CHAPTER 7

IMPLEMENTING ROLES

Overview	
Goal	Create and manage roles
Objectives	<ul style="list-style-type: none">• Describe the structure and behavior of a role• Create a role• Deploy roles with Ansible Galaxy
Sections	<ul style="list-style-type: none">• Describing Role Structure (and Quiz)• Creating Roles (and Guided Exercise)• Deploying Roles with Ansible Galaxy (and Guided Exercise)
Lab	<ul style="list-style-type: none">• Implementing Roles

Describing Role Structure

Objectives

After completing this section, students should be able to:

- Describe the structure and behavior of a role.
- Define role dependencies.

Structuring Ansible playbooks with roles

Data centers have a variety of different types of hosts. Some serve as web servers, others as database servers, and others can have software development tools installed and configured on them. An Ansible playbook, with tasks and handlers to handle all of these cases, would become large and complex over time. Ansible *roles* allow administrators to organize their playbooks into separate, smaller playbooks and files.

Roles provide Ansible with a way to load tasks, handlers, and variables from external files. Static files and templates can also be associated and referenced by a role. The files that define a role have specific names and are organized in a rigid directory structure, which will be discussed later. Roles can be written so they are general purpose and can be reused.

Use of Ansible roles has the following benefits:

- Roles group content, allowing easy sharing of code with others
- Roles can be written that define the essential elements of a system type: web server, database server, git repository, or other purpose
- Roles make larger projects more manageable
- Roles can be developed in parallel by different administrators

Examining the Ansible role structure

An Ansible role's functionality is defined by its directory structure. The top-level directory defines the name of the role itself. Some of the subdirectories contain YAML files, named **main.yml**. The **files** and **templates** subdirectories can contain objects referenced by the YAML files.

The following **tree** command displays the directory structure of the **user.example** role.

```
[user@host roles]$ tree user.example
user.example/
├── defaults
│   └── main.yml
├── files
├── handlers
│   └── main.yml
├── meta
│   └── main.yml
└── README.md

    ├── tasks
    │   └── main.yml
```

```

  templates
  tests
    inventory
      test.yml
  vars
    main.yml

```

Ansible role subdirectories

Subdirectory	Function
defaults	The main.yml file in this directory contains the default values of role variables that can be overwritten when the role is used.
files	This directory contains static files that are referenced by role tasks.
handlers	The main.yml file in this directory contains the role's handler definitions.
meta	The main.yml file in this directory contains information about the role, including author, license, platforms, and optional role dependencies.
tasks	The main.yml file in this directory contains the role's task definitions.
templates	This directory contains Jinja2 templates that are referenced by role tasks.
tests	This directory can contain an inventory and test.yml playbook that can be used to test the role.
vars	The main.yml file in this directory defines the role's variable values.

Defining variables and defaults

Role variables are defined by creating a **vars/main.yml** file with key:value pairs in the role directory hierarchy. They are referenced in the role YAML file like any other variable: **{{ VAR_NAME }}**. These variables have a high priority and can not be overridden by inventory variables.

Default variables allow default values to be set for variables of included or dependent roles. They are defined by creating a **defaults/main.yml** file with key:value pairs in the role directory hierarchy. Default variables have the lowest priority of any variables available. They can be easily overridden by any other variable, including inventory variables.

Define a specific variable in either **vars/main.yml** or **defaults/main.yml**, but not in both places. Default variables should be used when it is intended that their values will be overridden.

Using Ansible roles in a playbook

Using roles in a playbook are straightforward. The following example shows how to use Ansible roles.

```

---
- hosts: remote.example.com
  roles:
    - role1
    - role2

```

For each role specified, the role tasks, role handlers, role variables, and role dependencies will be included in the playbook, in that order. Any **copy**, **script**, **template**, or **include** tasks in the role can reference the relevant files, templates, or tasks without absolute or relative path names.

Ansible will look for them in the role's **files**, **templates**, or **tasks** respectively, based on their use.

The following example shows the alternative syntax for using a role in a playbook. **role1** is used in the same way as the previous example. Default variable values are overridden when **role2** is used.

```
---  
- hosts: remote.example.com  
  roles:  
    - role: role1  
    - role: role2  
      var1: val1  
      var2: val2
```

Defining role dependencies

Role dependencies allow a role to include other roles as dependencies in a playbook. For example, a role that defines a documentation server may depend upon another role that installs and configures a web server. Dependencies are defined in the **meta/main.yml** file in the role directory hierarchy.

The following is a sample **meta/main.yml** file.

```
---  
dependencies:  
  - { role: apache, port: 8080 }  
  - { role: postgres, dbname: serverlist, admin_user: felix }
```

By default, roles are only added as a dependency to a playbook once. If another role also lists it as a dependency it will not be run again. This behavior can be overridden by setting the **allow_duplicates** variable to **yes** in the **meta/main.yml** file.

Controlling order of execution

Normally, the tasks of roles execute before the tasks of the playbooks that use them. Ansible provides a way of overriding this default behavior: the **pre_tasks** and **post_tasks** tasks.

The **pre_tasks** tasks are performed before any roles are applied. The **post_tasks** tasks are performed after all the roles have completed.

```
---  
- hosts: remote.example.com  
  pre_tasks:  
    - debug:  
      msg: 'hello'  
  roles:  
    - role1  
    - role2  
  tasks:  
    - debug:  
      msg: 'still busy'  
  post_tasks:  
    - debug:  
      msg: 'goodbye'
```



References

Playbook Roles and Include Statements – Ansible Documentation
http://docs.ansible.com/ansible/playbooks_roles.html

Quiz: Describing Role Structure

Choose the correct answer to the following questions:

1. Roles are:
 - a. Configuration settings that allow specific users to run Ansible playbooks.
 - b. Playbooks for a data center.
 - c. Collection of YAML task files and supporting items arranged in a specific structure for easy sharing, portability, and reuse.
2. Which of the following can be specified in roles?
 - a. Handlers
 - b. Tasks
 - c. Templates
 - d. Variables
 - e. All of the above
3. Which file declares role dependencies?
 - a. The Ansible playbook that uses the role.
 - b. The **meta/main.yml** file inside the role hierarchy.
 - c. The **meta/main.yml** file in the project directory.
 - d. Role dependencies cannot be defined in Ansible.
4. Which file in a role's directory hierarchy should contain the initial values of variables that might be used as parameters to the role?
 - a. **defaults/main.yml**
 - b. **meta/main.yml**
 - c. **vars/main.yml**
 - d. The host inventory file.

Solution

Choose the correct answer to the following questions:

1. Roles are:
 - a. Configuration settings that allow specific users to run Ansible playbooks.
 - b. Playbooks for a data center.
 - c. **Collection of YAML task files and supporting items arranged in a specific structure for easy sharing, portability, and reuse.**
2. Which of the following can be specified in roles?
 - a. Handlers
 - b. Tasks
 - c. Templates
 - d. Variables
 - e. **All of the above**
3. Which file declares role dependencies?
 - a. The Ansible playbook that uses the role.
 - b. **The meta/main.yml file inside the role hierarchy.**
 - c. The meta/main.yml file in the project directory.
 - d. Role dependencies cannot be defined in Ansible.
4. Which file in a role's directory hierarchy should contain the initial values of variables that might be used as parameters to the role?
 - a. **defaults/main.yml**
 - b. meta/main.yml
 - c. vars/main.yml
 - d. The host inventory file.

Creating Roles

Objectives

After completing this section, students should be able to:

- Create an Ansible role.
- Properly reference an Ansible role in a playbook.

Creating roles in Ansible requires no special development tools. Creating and using a role is a three step process:

1. Create the role directory structure.
2. Define the role content.
3. Use the role in a playbook.

Creating the role directory structure

Ansible looks for roles in a subdirectory called **roles** in the project directory. Roles can also be kept in the directories referenced by the **roles_path** variable in Ansible configuration files. This variable contains a colon-separated list of directories to search.

Each role has its own directory with specially named subdirectories. The following directory structure contains the files that define the **motd** role.

```
[user@host ~]$ tree roles/
roles/
└── motd
    ├── defaults
    │   └── main.yml
    ├── files
    ├── handlers
    ├── tasks
    │   └── main.yml
    └── templates
        └── motd.j2
```

The **files** subdirectory contains fixed-content files and the **templates** subdirectory contains templates that can be deployed by the role when it is used. The other subdirectories can contain **main.yml** files that define default variable values, handlers, tasks, role metadata, or variables, depending on the subdirectory they are in. If a subdirectory exists but is empty, such as **handlers** in the previous example, it is ignored. If a role does not utilize a feature, the subdirectory can be omitted altogether, for example the **meta** and **vars** subdirectories in the previous example.

Defining the role content

After the directory structure is created, the content of the Ansible role must be defined. A good place to start would be the **ROLENAME/tasks/main.yml** file. This file defines which modules to call on the managed hosts that this role is applied.

The following **tasks/main.yml** file manages the **/etc/motd** file on managed hosts. It uses the **template** module to copy the template named **motd.j2** to the managed host. The template is retrieved from the **templates** subdirectory of the role.

```
[user@host ~]$ cat roles/motd/tasks/main.yml
---
# tasks file for motd

- name: deliver motd file
  template:
    src: templates/motd.j2
    dest: /etc/motd
    owner: root
    group: root
    mode: 0444
```

The following command displays the contents of the **templates/motd.j2** template of the **motd** role. It references Ansible facts and a **system-owner** variable.

```
[user@host ~]$ cat roles/motd/templates/motd.j2
This is the system {{ ansible_hostname }}.

Today's date is: {{ ansible_date_time.date }}.

Only use this system with permission.
You can ask {{ system_owner }} for access.
```

The role can define a default value for the **system_owner** variable. The **defaults/main.yml** file in the role's directory structure is where these values are set.

The following **defaults/main.yml** file sets the **system_owner** variable to **user@host.example.com**. This will be the email address that is written in the **/etc/motd** file of managed hosts that this role is applied to.

```
[user@host ~]$ cat roles/motd/defaults/main.yml
---
system_owner: user@host.example.com
```

Using the role in a playbook

To access a role, reference it in the **roles:** section of a playbook. The following playbook refers to the **motd** role. Because no variables are specified, the role will be applied with its default variable values.

```
[user@host ~]$ cat use-motd-role.yml
---
- name: use motd role playbook
  hosts: remote.example.com
  user: devops
  become: true

  roles:
    - motd
```

When the playbook is executed, tasks performed because of a role can be identified by the role name prefix. The following sample output illustrates this with the **motd: deliver motd file** message.

```
[user@host ~]$ ansible-playbook -i inventory use-motd-role.yml
PLAY [use motd role playbook] ****
TASK [setup] ****
ok: [remote.example.com]

TASK [motd : deliver motd file] ****
changed: [remote.example.com]

PLAY RECAP ****
remote.example.com      : ok=2    changed=1    unreachable=0    failed=0
```

Changing a role's behavior with variables

Variables can be used with roles, like parameters, to override previously defined, default values. When they are referenced, the variable:value pairs must be specified as well.

The following example shows how to use the **motd** role with a different value for the **system_owner** role variable. The value specified, **someone@host.example.com**, will replace the variable reference when the role is applied to a managed host.

```
[user@host ~]$ cat use-motd-role.yml
---
- name: use motd role playbook
  hosts: remote.example.com
  user: devops
  become: true

  roles:
    - role: motd
      system_owner: someone@host.example.com
```

References

Playbook Roles and Include Statements – Ansible Documentation
http://docs.ansible.com/ansible/playbooks_roles.html

Guided Exercise: Creating Roles

In this exercise, you will create two roles that use variables and parameters: **myvhost** and **myfirewall**.

The **myvhost** role will install and configure the Apache service on a host. A template is provided that will be used for **/etc/httpd/conf.d/vhost.conf: vhost.conf.j2**.

The **myfirewall** role installs, enables, and starts the **firewalld** daemon. It opens the firewall service port specified by the **firewall_service** variable.

Outcomes

You should be able to create Ansible roles that use variables, files, templates, tasks, and handlers to deploy a network service and enable a working firewall.

Before you begin

Reset **servera**.

From **workstation**, run the command **lab creating-roles setup** to prepare the environment for this exercise. This will create the working directory, **dev-roles**, and populate it with an Ansible configuration file and host inventory.

```
[student@workstation ~]$ lab creating-roles setup
```

Steps

1. Log in to your **workstation** host as **student**. Change to the **dev-roles** working directory.

```
[student@workstation ~]$ cd ~/dev-roles  
[student@workstation dev-roles]$
```

2. Create the directory structure for a role called **myvhost**. The role will include fixed files, templates, tasks, and handlers. A dependency will be created later, so it should have a **meta** subdirectory.

```
[student@workstation dev-roles]$ mkdir -p roles/myvhost/{files,handlers}  
[student@workstation dev-roles]$ mkdir roles/myvhost/{meta,tasks,templates}
```

3. Create the **main.yml** file in the **tasks** subdirectory of the role. The role should perform four tasks:

- Install the **httpd** package.
 - Start and enable the **httpd** service.
 - Download the HTML content into the virtual host **DocumentRoot** directory.
 - Install the template configuration file that configures the webserver.
- 3.1. Use a text editor to create a file called **roles/myvhost/tasks/main.yml**. Include code to use the **yum** module to install the **httpd** package. The file contents should look like the following:

```
---  
# tasks file for myvhost  
  
- name: install httpd  
  yum:  
    name: httpd  
    state: latest
```

- 3.2. Add additional code to the **tasks/main.yml** file to use the **service** module to start and enable the **httpd** service.

```
- name: start and enable httpd service  
  service:  
    name: httpd  
    state: started  
    enabled: true
```

- 3.3. Add another stanza to copy the HTML content from the role to the virtual host **DocumentRoot** directory. Use the **copy** module and include a trailing slash after the source directory name. This will cause the module to copy the contents of the **html** directory immediately below the destination directory (similar to **rsync** usage). The **ansible_hostname** variable will expand to the short host name of the managed host.

```
- name: deliver html content  
  copy:  
    src: html/  
    dest: "/var/www/vhosts/{{ ansible_hostname }}/"
```

- 3.4. Add another stanza to use the **template** module to create **/etc/httpd/conf.d/vhost.conf** on the managed host. It should call a handler to restart the **httpd** daemon when this file is updated.

```
- name: template vhost file  
  template:  
    src: vhost.conf.j2  
    dest: /etc/httpd/conf.d/vhost.conf  
    owner: root  
    group: root  
    mode: 0644  
  notify:  
    - restart httpd
```

- 3.5. Save your changes and exit the **tasks/main.yml** file.

4. Create the handler for restarting the **httpd** service. Use a text editor to create a file called **roles/myvhost/handlers/main.yml**. Include code to use the **service** module. The file contents should look like the following:

```
---  
# handlers file for myvhost  
  
- name: restart httpd  
  service:  
    name: httpd
```

```
state: restarted
```

5. Create the HTML content that will be served by the webserver.
 - 5.1. The role task that called the **copy** module referred to an **html** directory as the **src**. Create this directory below the **files** subdirectory of the role.

```
[student@workstation dev-roles]$ mkdir -p roles/myvhost/files/html
```

- 5.2. Create an **index.html** file below that directory with the contents: "simple index". Be sure to use this string verbatim because the grading script looks for it.
- ```
[student@workstation dev-roles]$ echo 'simple index' > roles/myvhost/files/html/index.html
```
6. Move the **vhost.conf.j2** template from the project directory to the role's **templates** subdirectory.

```
[student@workstation dev-roles]$ mv vhost.conf.j2 roles/myvhost/templates/
```

7. Test the **myvhost** role to make sure it works properly.
  - 7.1. Write a playbook that uses the role, called **use-vhost-role.yml**. It should have the following content:

```

- name: use vhost role playbook
 hosts: webservers

 pre_tasks:
 - debug:
 msg: 'Beginning web server configuration.'

 roles:
 - myvhost

 post_tasks:
 - debug:
 msg: 'Web server has been configured.'
```

- 7.2. Before running the playbook, verify its syntax is correct by running **ansible-playbook --syntax-check**. If it reports any errors, correct them before moving to the next step. You should see output similar to the following:

```
[student@workstation dev-roles]$ ansible-playbook --syntax-check use-vhost-role.yml

playbook: use-vhost-role.yml
```

- 7.3. Run the playbook. Review the output to confirm that Ansible performed the actions on the web server, **servera**.

```
[student@workstation dev-roles]$ ansible-playbook use-vhost-role.yml

PLAY [use vhost role playbook] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [debug] ****
ok: [servera.lab.example.com] => {
 "msg": "Beginning web server configuration."
}

TASK [myvhost : install httpd] ****
changed: [servera.lab.example.com]

TASK [myvhost : start and enable httpd service] ****
changed: [servera.lab.example.com]

TASK [myvhost : deliver html content] ****
changed: [servera.lab.example.com]

TASK [myvhost : template vhost file] ****
changed: [servera.lab.example.com]

RUNNING HANDLER [myvhost : restart httpd] ****
changed: [servera.lab.example.com]

TASK [debug] ****
ok: [servera.lab.example.com] => {
 "msg": "Web server has been configured."
}

PLAY RECAP ****
servera.lab.example.com : ok=8 changed=5 unreachable=0 failed=0
```

- 7.4. Run ad hoc commands to confirm that the role worked. The *httpd* package should be installed and the **httpd** service should be running.

```
[student@workstation dev-roles]$ ansible webservers -a 'yum list installed
httpd'
servera.lab.example.com | SUCCESS | rc=0 >>
Loaded plugins: langpacks, search-disabled-repos
Installed Packages
httpd.x86_64 2.4.6-45.el7 @rhel_dvd
[student@workstation dev-roles]$ ansible webservers -a 'systemctl is-active
httpd'
servera.lab.example.com | SUCCESS | rc=0 >>
active
[student@workstation dev-roles]$ ansible webservers -a 'systemctl is-enabled
httpd'
servera.lab.example.com | SUCCESS | rc=0 >>
enabled
```

- 7.5. The Apache configuration should be installed with template variables expanded.

```
[student@workstation dev-roles]$ ansible webservers -a 'cat /etc/httpd/conf.d/
vhost.conf'
servera.lab.example.com | SUCCESS | rc=0 >>
Ansible managed: /home/student/dev-roles/roles/myvhost/templates/vhost.conf.j2
```

```
modified on 2016-04-15 10:01:12 by student on workstation.lab.example.com

<VirtualHost *:80>
 ServerAdmin webmaster@servera.lab.example.com
 ServerName servera.lab.example.com
 ErrorLog logs/servera-error.log
 CustomLog logs/servera-common.log common
 DocumentRoot /var/www/vhosts/servera/

 <Directory /var/www/vhosts/servera/>
 Options +Indexes +FollowSymlinks +Includes
 Order allow,deny
 Allow from all
 </Directory>
</VirtualHost>
```

- 7.6. The HTML content should be found in a directory called **/var/www/vhosts/servera**. The **index.html** file should contain the string "simple index".

```
[student@workstation dev-roles]$ ansible webservers -a 'cat /var/www/vhosts/
servera/index.html'
servera.lab.example.com | SUCCESS | rc=0 >>
simple index
```

- 7.7. Use a web browser on **servera** to check if the web content is available locally. It should succeed and display content.

```
[student@workstation dev-roles]$ ansible webservers -a 'curl -s http://
localhost'
servera.lab.example.com | SUCCESS | rc=0 >>
simple index
```

- 7.8. Use a web browser on **workstation** to check if content is available from **http://servera.lab.example.com**. If a firewall is running on **servera**, you will see the following error message:

```
[student@workstation dev-roles]$ curl -S http://servera.lab.example.com
curl: (7) Failed connect to servera.lab.example.com:80; No route to host
```

This is because the firewall port for HTTP is not open. If the web content successfully displays, it is because a firewall is not running on **servera**.

8. Create the role directory structure for a role called **myfirewall**.

```
[student@workstation dev-roles]$ mkdir -p roles/myfirewall/{defaults,handlers,tasks}
```

9. Create the **main.yml** file in the **tasks** subdirectory of the role. The role should perform three tasks:
- Install the **firewalld** package.
  - Start and enable the **firewalld** service.
  - Open a firewall service port.

- 9.1. Use a text editor to create a file called **roles/myfirewall/tasks/main.yml**. Include code to use the **yum** module to install the *firewalld* package. The file contents should look like the following:

```

tasks file for myfirewall

- name: install firewalld
 yum:
 name: firewalld
 state: latest
```

- 9.2. Add additional code to the **tasks/main.yml** file to use the **service** module to start and enable the **firewalld** service.

```
- name: start and enable firewalld service
 service:
 name: firewalld
 state: started
 enabled: true
```

- 9.3. Add another stanza to use the **firewalld** module to immediately, and persistently, open the service port specified by the **firewall\_service** variable. It should look like the following:

```
- name: firewall services config
 firewalld:
 state: enabled
 immediate: true
 permanent: true
 service: "{{ firewall_service }}"
```

- 9.4. Save your changes and exit the **tasks/main.yml** file.

10. Create the handler for restarting the **firewalld** service. Use a text editor to create a file called **roles/myfirewall/handlers/main.yml**. Include code to use the **service** module. The file contents should look like the following:

```

handlers file for myfirewall

- name: restart firewalld
 service:
 name: firewalld
 state: restarted
```

11. Create the file that defines the default value for the **firewall\_service** variable. It should have a default value of **ssh** initially. We will override the value to open the port for **http** when we use the role in a later step.

Use a text editor to create a file called **roles/myfirewall/defaults/main.yml**. It should contain the following content:

```

defaults file for myfirewall

firewall_service: ssh
```

12. Modify the **myvhost** role to include the **myfirewall** role as a dependency, then retest the modified role.

- 12.1. Use a text editor to create a file, called **roles/myvhost/meta/main.yml**, that makes **myvhost** depend on the **myfirewall** role. The **firewall\_service** variable should be set to **http** so the correct service port is opened. By using the role in this way, the default value of **ssh** for **firewall\_service** will be ignored. The explicitly assigned value of **http** will be used instead.

The resulting file should look like the following:

```

dependencies:
- { role: myfirewall, firewall_service: http }
```

- 12.2. Run the playbook again. Confirm the additional **myfirewall** tasks are successfully executed.

```
[student@workstation dev-roles]$ ansible-playbook use-vhost-role.yml

PLAY [use vhost role playbook] ****
... Output omitted ...

TASK [myfirewall : install firewalld] ****
ok: [servera.lab.example.com]

TASK [myfirewall : start and enable firewalld daemon] ****
ok: [servera.lab.example.com]

TASK [myfirewall : firewall services config] ****
changed: [servera.lab.example.com]

... Output omitted ...

PLAY RECAP ****
servera.lab.example.com : ok=10 changed=6 unreachable=0 failed=0
```

- 12.3. Confirm the web server content is available to remote clients.

```
[student@workstation dev-roles]$ curl http://servera.lab.example.com
simple index
```

### Evaluation

From **workstation**, run the **lab creating-roles grade** command to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab creating-roles grade
```

#### Cleanup

Run the **lab creating-roles cleanup** command to cleanup the managed host.

```
[student@workstation ~]$ lab creating-roles cleanup
```

# Deploying Roles with Ansible Galaxy

## Objectives

After completing this section, students should be able to:

- Locate Ansible roles in the Ansible Galaxy website.
- Deploy roles with Ansible Galaxy.

## Ansible Galaxy

*Ansible Galaxy* [<https://galaxy.ansible.com>] is a public library of Ansible roles written by a variety of Ansible administrators and users. It is an archive that contains thousands of Ansible roles and it has a searchable database that helps Ansible users identify roles that might help them accomplish an administrative task. Ansible Galaxy includes links to documentation and videos for new Ansible users and role developers.

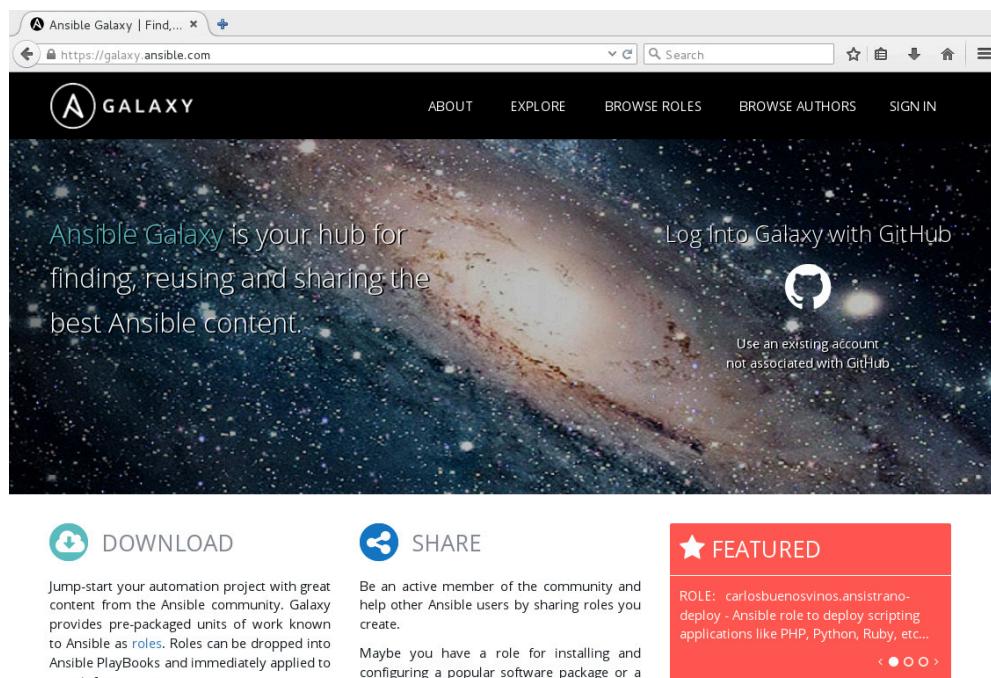


Figure 7.1: Ansible Galaxy home page

### Accessing tips for using Ansible Galaxy

The **ABOUT** tab on the Ansible Galaxy website home page leads to a page that describes how to use Ansible Galaxy. There is content that describes how to download and use roles from Ansible Galaxy. Instructions on how to develop roles and upload them to Ansible Galaxy are also on that page.

There are counters associated with each role that is found on the Ansible Galaxy website. Users can vote for the usefulness of a role by clicking the star button. The number of stars a role has is a clue to its popularity among the Ansible community. Users can also watch a role. The number of watchers for a role give an indication of community interest in a role under development. Finally,

the number of times a role is downloaded from Ansible Galaxy is maintained with the role. This count is an indication of how many Ansible users actually use the role.

The **EXPLORE** tab on the Ansible Galaxy website home page shows the most active, the most starred, and the most watched roles on Ansible Galaxy. This page also displays the list of the top role authors and contributors on the site.

### Browsing Ansible roles and authors

The **BROWSE ROLES** and **BROWSE AUTHORS** tabs on the Ansible Galaxy website home page give users access to information about the roles published on Ansible Galaxy. Users can search for an Ansible role by its name, or by other role attributes. The authors of roles published in Ansible Galaxy can be searched by name.

The following figure shows the search results that displayed after a keyword search was performed. The term “install git” was entered into the **Search roles** box.

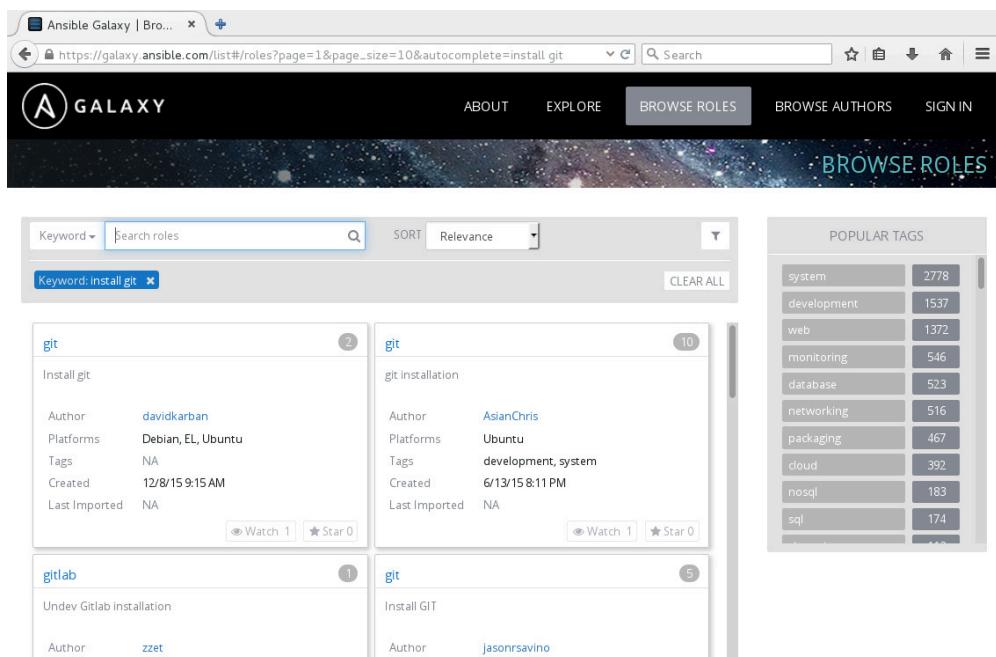


Figure 7.2: Ansible Galaxy search screen

The pulldown menu to the left of the **Search roles** box allow searches to be performed on keywords, author IDs, platform, and tags. Possible platform values include **EL** for Enterprise Linux, and **Fedora**. Tags are often assigned to roles to indicate their use in the data center. Possible tag values include **system**, **development**, **web**, **packaging**, and others.

## The ansible-galaxy command-line tool

The **ansible-galaxy** command line tool can be used to search for, display information about, install, list, remove, or initialize roles.

### Identifying and installing roles

The **ansible-galaxy search** subcommand searches Ansible Galaxy for the string specified as an argument. The **--author**, **--platforms**, and **--galaxy-tags** options can be used to narrow the search results. The following example displays the names of roles that include “install” and “git” in their description, and are available for the Enterprise Linux (**e1**) platform.

```
[user@host ~]$ ansible-galaxy search 'install git' --platforms el

Found 77 roles matching your search:

 Name Description

 zzet.gitlab Undev Gitlab installation
 jasonrsavino.git Install GIT
 samdoran.gitlab Install GitLab CE Omnibus
 kbrebanov.git Installs git
 AsianChris.git git installation
 ... Output omitted ...
```

The **ansible-galaxy info** subcommand displays more detailed information about a role. The following command displays information about the **davidkarban.git** role, available from Ansible Galaxy. Because the information requires more than one screen to display, **ansible-galaxy** uses **less** to display the role's information.

```
[user@host ~]$ ansible-galaxy info davidkarban.git
[DEPRECATION WARNING]: The comma separated role spec format, use the
yaml/explicit format instead.. This feature will be removed in a future release.
 Deprecation warnings can be disabled by setting deprecation_warnings=False in
 ansible.cfg.
less 458 (POSIX regular expressions)
Copyright (C) 1984-2012 Mark Nudelman

less comes with NO WARRANTY, to the extent permitted by law.
For information about the terms of redistribution,
see the file named README in the less distribution.
Homepage: http://www.greenwoodsoftware.com/less

Role: davidkarban.git
 description: Install git
 active: True
 commit:
 commit_message:
 commit_url:
 company: David Karban
 created: 2015-12-08T09:15:48.542Z
 download_count: 1
 forks_count: 0
 github_branch:
 github_repo: ansible-git
 github_user: davidkarban
 id: 6422
 is_valid: True
 issue_tracker_url: https://github.com/davidkarban/ansible-git/issues
 license: license (GPLv2, CC-BY, etc)
 min_ansible_version: 1.2
 modified: 2016-04-20T20:13:35.549Z
 namespace: davidkarban
 open_issues_count: 0
 path: /etc/ansible/roles
:
```

The **ansible-galaxy install** subcommand downloads a role from Ansible Galaxy, then installs it locally on the control node. The default installation location for roles is **/etc/ansible/roles**. This location can be overridden by either the value of the **role\_path** configuration variable, or a **-p DIRECTORY** option on the command-line.

```
[user@host ~]$ ansible-galaxy install davidkarban.git -p roles/
- downloading role 'git', owned by davidkarban
- downloading role from https://github.com/davidkarban/ansible-git/archive/master.tar.gz
- extracting davidkarban.git to roles/davidkarban.git
- davidkarban.git was installed successfully
[user@host ~]$ ls roles/
davidkarban.git
```

Multiple roles can be installed with a single **ansible-galaxy install** command when a requirements file is specified with the **-r** option and the destination directory with the **-p** option. The requirements file is a YAML file that specifies which roles to download and install. A **name** value can be used to determine what the role will be called locally.

There are many types of role sources that can be specified in a requirements file. The following example shows how to specify a download from Ansible Galaxy and another download from another web server.

```
[user@host ~]$ cat roles2install.yml
From Galaxy
- src: author.rolename

From a webserver, where the role is packaged in a gzipped tar archive
- src: https://webserver.example.com/files/sample.tgz
 name: ftpserver-role
[user@host ~]$ ansible-galaxy install -r roles2install.yml -p roles-directory
```

Roles downloaded and installed from Ansible Galaxy can be used in playbooks like any other role. They are referenced in the **roles:** section using their full **AUTHOR.NAME** role name. The following **use-git-role.yml** playbook references the **davidkarban.git** role.

```
[user@host ~]$ cat use-git-role.yml

- name: use davidkarban.git role playbook
 hosts: remote.example.com
 user: devops
 become: true

 roles:
 - davidkarban.git
```

Using the playbook causes *git* to be installed on the **remote.example.com** server. The full role name, including the author, is displayed as its tasks are executed by the playbook.

```
[user@host ~]$ ssh remote.example.com yum list installed git
Loaded plugins: langpacks, search-disabled-repos
Error: No matching Packages to list
[user@host ~]$ ansible-playbook use-git-role.yml

PLAY [use davidkarban.git role playbook] ****
TASK [setup] ****
ok: [remote.example.com]

TASK [davidkarban.git : Load the OS specific variables] ****
ok: [remote.example.com]

TASK [davidkarban.git : Install the packages in Redhat derivates] ****
```

```
... Output omitted ...

PLAY RECAP ****
remote.example.com : ok=3 changed=1 unreachable=0 failed=0

[user@host ~]$ ssh remote.example.com yum list installed git
Loaded plugins: langpacks, search-disabled-repos
Installed Packages
git.x86_64 1.8.3.1-5.el7 @rhel_dvd
```

### Managing downloaded modules

The **ansible-galaxy** command can manage local roles. The roles are found in the **roles** directory of the current project, or they can be found in one of the directories listed in the **roles\_path** variable. The **ansible-galaxy list** subcommand lists the roles that are found locally.

```
[user@host ~]$ ansible-galaxy list
- davidkarban.git, master
- student.bash_env, (unknown version)
```

A role can be removed locally with the **ansible-galaxy remove** subcommand.

```
[user@host ~]$ ansible-galaxy remove student.bash_env
- successfully removed student.bash_env
[user@host ~]$ ansible-galaxy list
- davidkarban.git, master
```

### Using ansible-galaxy to create roles

The **ansible-galaxy init** command creates a directory structure for a new role that will be developed. The author and name of the role is specified as an argument to the command and it creates the directory structure in the current directory. **ansible-galaxy** interacts with the Ansible Galaxy website API when it performs most operations. The **--offline** option permits the **init** command to be used when Internet access is unavailable.



### Important

Depending on the Ansible version you are using, you might encounter a situation where **ansible-galaxy** command does not create all the required subdirectories. For example: **files** and **templates** might be missing. If this is the case simply create those two directories manually using **mkdir** command.

```
[user@host roles]$ ansible-galaxy init --offline student.example
- student.example was created successfully
[user@host roles]$ ls student.example/
defaults files handlers meta README.md tasks templates tests vars
```



## References

Ansible Galaxy – Ansible Documentation  
<http://docs.ansible.com/ansible/galaxy.html>

# Guided Exercise: Deploying Roles with Ansible Galaxy

In this exercise, you will use Ansible Galaxy to download and install an Ansible role. You will also use it to initialize the directory for a new role to be developed.

## Outcomes

You should be able to use Ansible Galaxy to initialize a new Ansible role and download and install an existing role.

## Before you begin

From workstation, run the command **lab ansible-galaxy setup** to prepare the environment for this exercise. This will create the working directory, **dev-roles**, and populate it with an Ansible configuration file and host inventory.

```
[student@workstation ~]$ lab ansible-galaxy setup
```

## Steps

1. Log in to your **workstation** host as **student**. Change to the **dev-roles** working directory.

```
[student@workstation ~]$ cd ~/dev-roles
[student@workstation dev-roles]$
```

2. Launch your favorite text editor and create a requirements file, called **install-roles.yml**. It should contain the following content:

```

install-roles.yml

- src: http://materials.example.com/roles-library/student.bash_env.tgz
 name: student.bash_env
```

The **src** value specifies the URL where the existing Ansible role exists. The **name** value defines where to save the role locally.

3. Use the **ansible-galaxy** command to utilize the requirements file you just created to download and install the **student.bash\_env** role.
  - 3.1. For comparison, display the contents of the **roles** subdirectory before the role is installed.

```
[student@workstation dev-roles]$ ls roles/
myfirewall myvhost
```

- 3.2. Use Ansible Galaxy to download and install the role. The **-p roles** option specifies the path to the directory where roles are stored locally. The **-r install-roles.yml** option specifies the requirements file listing the roles to download and install.

```
[student@workstation dev-roles]$ ansible-galaxy install -p roles -r install-roles.yml
- downloading role from http://materials.example.com/roles-library/
student.bash_env.tgz
- extracting student.bash_env to roles/student.bash_env
- student.bash_env was installed successfully
```

- 3.3. Display the **roles** subdirectory after the role has been installed. Confirm it has a new subdirectory, called **student.bash\_env**, matching the **name** value specified in the YAML file.

```
[student@workstation dev-roles]$ ls roles/
myfirewall myvhost student.bash_env
```

4. Create a playbook, called **use-bash\_env-role.yml**, that uses the **student.bash\_env** role. It should execute on the **webservers** host group as the **devops** user. The contents of the playbook should look like the following:

```

- name: use student.bash_env role playbook
 hosts: webservers
 user: devops
 become: true

 roles:
 - student.bash_env
```

5. Run the playbook. The **student.bash\_env** role creates standard template configuration files in **/etc/skel** on the managed host. The files it creates include **.bashrc**, **.bash\_profile**, and **.vimrc**.

```
[student@workstation dev-roles]$ ansible-playbook -i inventory use-bash_env-role.yml

PLAY [use student.bash_env role playbook] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [student.bash_env : put away .bashrc] ****
changed: [servera.lab.example.com]

TASK [student.bash_env : put away .bash_profile] ****
ok: [servera.lab.example.com]

TASK [student.bash_env : put away .vimrc] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com : ok=4 changed=2 unreachable=0 failed=0
```

6. Compare the template files in the **student.bash\_env** role with the files on the managed host. They will differ when the templates have variables that will be expanded on the managed host.

- 6.1. Use the **md5sum** command to generate checksums of the templates.

```
[student@workstation dev-roles]$ md5sum roles/student.bash_env/templates/*
f939eb71a81a9da364410b799e817202 roles/student.bash_env/templates/
_bash_profile.j2
989764ae6830691e7468599db8d194ba roles/student.bash_env/templates/_bashrc.j2
a7320d70317cd23e0c2083489b532cdf roles/student.bash_env/templates/_vimrc.j2
```

- 6.2. Display the MD5 checksums of the files on the managed host, **servera**, for comparison.

```
[student@workstation dev-roles]$ ssh servera md5sum /etc/skel/
{.bash_profile,.bashrc,.vimrc}
f939eb71a81a9da364410b799e817202 /etc/skel/.bash_profile
7f6d35286702531c9bef441516334404 /etc/skel/.bashrc
a7320d70317cd23e0c2083489b532cdf /etc/skel/.vimrc
```

The sums are the same for **.bash\_profile** and **.vimrc**.

- 6.3. Display the contents of the **\_bashrc.j2** template. It uses a variable for some of its content, and that is why it is different from the file on the managed host.

```
[student@workstation dev-roles]$ cat roles/student.bash_env/templates/_bashrc.j2
#
.bashrc

Source global definitions
if [-f /etc/bashrc]; then
 . /etc/bashrc
fi

Uncomment the following line if you don't like systemctl's auto-paging
feature:
export SYSTEMD_PAGER=

User specific aliases and functions

PS1="{{ default_prompt }}"
```

- 6.4. Display the last few lines of the **/etc/skel/.bashrc** file on **servera**. You should see that it has a line that defines a default **PS1** prompt that came from the template.

```
[student@workstation dev-roles]$ ssh servera tail -n5 /etc/skel/.bashrc
#
User specific aliases and functions

PS1=[student prompt \w]\$ "
```

7. The **ansible-galaxy** command can be used to initialize the subdirectories for a new role. When creating a new role, instead of creating the directory structure by hand, **ansible-galaxy init --offline** will do the work for you.
- 7.1. Use the **ansible-galaxy** command with the **init** subcommand to create a new role called **empty.example**. The **-p roles** option specifies the path to the directory where roles are stored locally.

```
[student@workstation dev-roles]$ ansible-galaxy init --offline -p roles/empty.example
- empty.example was created successfully
```

- 7.2. A new subdirectory is created in the **roles** directory for the new role that is being defined.

```
[student@workstation dev-roles]$ ls roles/
empty.example myfirewall myvhost student.bash_env
```

- 7.3. List the directory that was created for the new role. It has all of the subdirectories that could be used in a role definition.

```
[student@workstation dev-roles]$ ls roles/empty.example/
defaults files handlers meta README.md tasks templates tests vars
[student@workstation dev-roles]$ ls roles/empty.example/*
roles/empty.example/README.md

roles/empty.example/defaults:
main.yml

roles/empty.example/files:

roles/empty.example/handlers:
main.yml

roles/empty.example/meta:
main.yml

roles/empty.example/tasks:
main.yml

roles/empty.example/templates:

roles/empty.example/tests:
inventory test.yml

roles/empty.example/vars:
main.yml
```

- 7.4. Display the contents of the **tasks/main.yml** file of the new role. Notice that it is a simple YAML stub with a comment with the role's name. All of the YAML files created by Ansible Galaxy contain similar content.

```
[student@workstation dev-roles]$ cat roles/empty.example/tasks/main.yml

tasks file for empty.example
```

### Evaluation

From **workstation**, run the **lab ansible-galaxy grade** command to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab ansible-galaxy grade
```

---

### Cleanup

Run the **lab ansible-galaxy cleanup** command to cleanup the managed host.

```
[student@workstation ~]$ lab ansible-galaxy cleanup
```

# Lab: Implementing Roles

In this lab, you will create two roles that use variables and parameters: **student.myenv** and **myapache**.

The **student.myenv** role customizes a system with required packages and a helpful script for all users. It will customize a user account, specified by the **myenv\_user** variable, with a profile picture and an extra command alias in their `~/.bashrc` file.

The **myapache** role will install and configure the Apache service on a host. Two templates are provided which will be used for the `/etc/httpd/conf/httpd.conf` and the `/var/www/html/index.html` files: `apache_httpdconf.j2` and `apache_indexhtml.j2`, respectively.

## Outcomes

You should be able to create Ansible roles that use variables, files, templates, tasks, and handlers to customize user environments and deploy a network service.

### Before you begin

Prepare your systems for this exercise by running the **lab ansible-roles-lab setup** command on your **workstation** system. This will create the working directory, called **lab-roles**, and populate it with an Ansible configuration file and host inventory.

```
[student@workstation ~]$ lab ansible-roles-lab setup
```

## Steps

1. Log in to your **workstation** host as **student**. Change to the **lab-roles** working directory.

```
[student@workstation ~]$ cd ~/lab-roles
[student@workstation lab-roles]$
```

2. Create directories to contain the Ansible roles. They should be contained in the **student.myenv** and **myapache** directories, below `~student/lab-roles/roles` on **workstation**.
3. Install the `mkcd.sh.j2` file as a template for the **student.myenv** role.
4. Make `/usr/share/icons/hicolor/48x48/apps/system-logo-icon.png` available as a static profile image for the **student.myenv** role. It will ultimately be called `profile.png` on the managed host in a user's home directory.
5. Define **student.myenv** role tasks to perform the following steps:
  - Install packages defined by the **myenv\_packages** variable.
  - Copy the standard profile picture to the user's home directory as `~/profile.png`. Use the **myenv\_user** variable for the user name.
  - Add the following line to the user's `~/.bashrc` file: `alias tree='tree -C'`. Use the **myenv\_user** variable for the user name. Hint: The `lineinfile` module might be well suited for this task.

- 
- Install the `mkcd.sh.j2` template as `/etc/profile.d/mkcd.sh`. It should have user:group ownership of `root:root` and have `-rw-r--r--` permissions.

The role should fail with an error message when the `myenv_user` variable is an empty string.

6. Define the `myenv_packages` variable for the `student.myenv` role so it contains the following packages: `git`, `tree`, and `vim-enhanced`.
7. Assign the empty string as the default value for the `myenv_user` variable.
8. Create a playbook, called `myenv.yml`, that runs on all hosts. It should use the `student.myenv` role, but do not set the `myenv_user` variable. Test the `student.myenv` role and confirm that it fails.
9. Update the `myenv.yml` playbook so that it uses the `student.myenv` role, setting the `myenv_user` variable to `student`. Test the `student.myenv` role and confirm that it works properly.
10. Install the Apache-related Jinja2 template files, in the `lab-roles` project directory, to the `myapache` role.
11. Create a handler that will restart the `httpd` service.
12. Define `myapache` role tasks to perform the following steps:
  - Install the `httpd` and `firewalld` packages.
  - Copy the `apache_httpdconf.j2` template to `/etc/httpd/conf/httpd.conf`. The target file should be owned by `root` with `-r--r--r--` permissions. Restart Apache using the handler created previously.
  - Copy the `apache_indexhtml.j2` template to `/var/www/html/index.html`. The target file should be owned by `root` with `-r--r--r--` permissions.
  - Start and enable the `httpd` and `firewalld` services.
  - Open port 80/tcp on the firewall.
13. Create the default variable values for the `myapache` role. The `apache_enable` variable should have a default value of `false`.
14. Create a playbook, called `apache.yml`, that runs on `serverb`. It should use the `myapache` role, but use the default value of the `apache_enable` variable. Test the `myapache` role and confirm that it installs the packages, but does not deploy the web server.
15. Modify the `apache.yml` playbook so that it uses the `myapache` role, setting the `apache_enable` variable to `true`. Test the `myapache` role and confirm that it works properly.

### Evaluation

Grade your work by running the **lab ansible-roles-lab grade** command from your **workstation** machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab ansible-roles-lab grade
```

### Cleanup

Run the **lab ansible-roles-lab cleanup** command to clean up the managed host.

```
[student@workstation ~]$ lab ansible-roles-lab cleanup
```

# Solution

In this lab, you will create two roles that use variables and parameters: **student.myenv** and **myapache**.

The **student.myenv** role customizes a system with required packages and a helpful script for all users. It will customize a user account, specified by the **myenv\_user** variable, with a profile picture and an extra command alias in their `~/.bashrc` file.

The **myapache** role will install and configure the Apache service on a host. Two templates are provided which will be used for the `/etc/httpd/conf/httpd.conf` and the `/var/www/html/index.html` files: `apache_httpdconf.j2` and `apache_indexhtml.j2`, respectively.

## Outcomes

You should be able to create Ansible roles that use variables, files, templates, tasks, and handlers to customize user environments and deploy a network service.

### Before you begin

Prepare your systems for this exercise by running the `lab ansible-roles-lab setup` command on your **workstation** system. This will create the working directory, called **lab-roles**, and populate it with an Ansible configuration file and host inventory.

```
[student@workstation ~]$ lab ansible-roles-lab setup
```

### Steps

1. Log in to your **workstation** host as **student**. Change to the **lab-roles** working directory.

```
[student@workstation ~]$ cd ~/lab-roles
[student@workstation lab-roles]$
```

2. Create directories to contain the Ansible roles. They should be contained in the **student.myenv** and **myapache** directories, below `~student/lab-roles/roles` on **workstation**.

The **ansible-galaxy** command can be used to create the subdirectories for the roles.



### Important

Depending on the Ansible version you are using, you might encounter a situation where **ansible-galaxy** command does not create two required subdirectories: **files** and **templates**. If this is the case, simply create those two directories manually using **mkdir** command.

```
[student@workstation lab-roles]$ ansible-galaxy init --offline -p roles
student.myenv
- student.myenv was created successfully
[student@workstation lab-roles]$ ansible-galaxy init --offline -p roles myapache
- myapache was created successfully
```

3. Install the `mkcd.sh.j2` file as a template for the **student.myenv** role.

The lab setup script copied the file to the **lab-roles** working directory. Move it to the **roles/student.myenv/templates/** subdirectory.

```
[student@workstation lab-roles]$ mv mkcd.sh.j2 roles/student.myenv/templates/
```

4. Make **/usr/share/icons/hicolor/48x48/apps/system-logo-icon.png** available as a static profile image for the **student.myenv** role. It will ultimately be called **profile.png** on the managed host in a user's home directory.

Copy the file, renaming it to **roles/student.myenv/files/profile.png**.

```
[student@workstation lab-roles]$ cp /usr/share/icons/hicolor/48x48/apps/system-logo-
icon.png roles/student.myenv/files/profile.png
```

5. Define **student.myenv** role tasks to perform the following steps:
  - Install packages defined by the **myenv\_packages** variable.
  - Copy the standard profile picture to the user's home directory as **~/profile.png**. Use the **myenv\_user** variable for the user name.
  - Add the following line to the user's **~/.bashrc** file: **alias tree='tree -C'**. Use the **myenv\_user** variable for the user name. Hint: The **lineinfile** module might be well suited for this task.
  - Install the **mkcd.sh.j2** template as **/etc/profile.d/mkcd.sh**. It should have user:group ownership of **root:root** and have **-rw-r--r--** permissions.

The role should fail with an error message when the **myenv\_user** variable is an empty string.

Modify **roles/student.myenv/tasks/main.yml** so that it contains the following:

```

tasks file for student.myenv

- name: check myenv_user default
 fail:
 msg: You must specify the variable `myenv_user` to use this role!
 when: "myenv_user == ''"

- name: install my packages
 yum:
 name: "{{ item }}"
 state: installed
 with_items: "{{ myenv_packages }}"

- name: copy placeholder profile pic
 copy:
 src: profile.png
 dest: "~{{ myenv_user }}/profile.png"

- name: set an alias in `~/.bashrc`
 lineinfile:
 line: "alias tree='tree -C'"
```

```

dest: "~{{ myenv_user }}/.bashrc"

- name: template out mkcd function
 template:
 src: mkcd.sh.j2
 dest: /etc/profile.d/mkcd.sh
 owner: root
 group: root
 mode: 0644

```

6. Define the **myenv\_packages** variable for the **student.myenv** role so it contains the following packages: *git*, *tree*, and *vim-enhanced*.

Create **roles/student.myenv/vars/main.yml** with the following contents:

```

vars file for student.myenv

myenv_packages:
 - 'git'
 - 'tree'
 - 'vim-enhanced'

```

7. Assign the empty string as the default value for the **myenv\_user** variable.

Create **roles/student.myenv/defaults/main.yml** with the following contents:

```

defaults file for student.myenv

myenv_user: ''

```

8. Create a playbook, called **myenv.yml**, that runs on all hosts. It should use the **student.myenv** role, but do not set the **myenv\_user** variable. Test the **student.myenv** role and confirm that it fails.

8.1. Use a text editor to create the **myenv.yml** playbook. It should look like the following:

```

- name: setup my personal environment
 hosts: all
 roles:
 - student.myenv

```

8.2. Run the **myenv.yml** playbook. Check the **ansible-playbook** output to make sure it fails.

```

[student@workstation lab-roles]$ ansible-playbook myenv.yml

PLAY [setup my personal environment] ****
TASK [Gathering Facts] ****
ok: [serverc.lab.example.com]
ok: [serverb.lab.example.com]

TASK [student.myenv : check myenv_user default] ****

```

```
fatal: [serverc.lab.example.com]: FAILED! => {"changed": false, "failed": true, "msg": "You must specify the variable `myenv_user` to use this role!"}
fatal: [serverb.lab.example.com]: FAILED! => {"changed": false, "failed": true, "msg": "You must specify the variable `myenv_user` to use this role!"}

 to retry, use: --limit @myenv.retry

PLAY RECAP *****
serverb.lab.example.com : ok=1 changed=0 unreachable=0 failed=1
serverc.lab.example.com : ok=1 changed=0 unreachable=0 failed=1
```

9. Update the **myenv.yml** playbook so that it uses the **student.myenv** role, setting the **myenv\_user** variable to **student**. Test the **student.myenv** role and confirm that it works properly.

- 9.1. Use a text editor to modify the **myenv.yml** playbook. It should look like the following:

```

- name: setup my personal environment
 hosts: all
 roles:
 - role: student.myenv
 myenv_user: student
```

- 9.2. Run the **myenv.yml** playbook. Check the **ansible-playbook** output to make sure the tasks ran properly.

```
[student@workstation lab-roles]$ ansible-playbook myenv.yml

PLAY [setup my personal environment] *****

TASK [Gathering Facts] *****
ok: [serverc.lab.example.com]
ok: [serverb.lab.example.com]

TASK [student.myenv : check myenv_user default] *****
skipping: [serverb.lab.example.com]
skipping: [serverc.lab.example.com]

TASK [student.myenv : install my packages] *****
changed: [serverb.lab.example.com] => (item=['vim-enhanced', 'tree', 'git'])
changed: [serverc.lab.example.com] => (item=['vim-enhanced', 'tree', 'git'])

TASK [student.myenv : copy placeholder profile pic] *****
changed: [serverb.lab.example.com]
changed: [serverc.lab.example.com]

TASK [student.myenv : set an alias in `~/.bashrc`] *****
changed: [serverc.lab.example.com]
changed: [serverb.lab.example.com]

TASK [student.myenv : template out mkcd function] *****
changed: [serverc.lab.example.com]
changed: [serverb.lab.example.com]

PLAY RECAP *****
serverb.lab.example.com : ok=5 changed=4 unreachable=0 failed=0
```

```
serverc.lab.example.com : ok=5 changed=4 unreachable=0 failed=0
```

10. Install the Apache-related Jinja2 template files, in the **lab-roles** project directory, to the **myapache** role.

- 10.1. The lab setup script installed the templates in the **lab-roles** working directory. Move them to the **roles/myapache/templates/** subdirectory.

```
[student@workstation lab-roles]$ mv apache_*.j2 roles/myapache/templates
```

- 10.2. List the **roles/myapache/templates/** subdirectory to confirm the templates are in place.

```
[student@workstation lab-roles]$ ls roles/myapache/templates
apache_httpdconf.j2 apache_indexhtml.j2
```

11. Create a handler that will restart the **httpd** service.

Modify **roles/myapache/handlers/main.yml** so that it contains the following:

```

handlers file for myapache

- name: restart apache
 service:
 name: httpd
 state: restarted
```

12. Define **myapache** role tasks to perform the following steps:

- Install the *httpd* and *firewalld* packages.
- Copy the **apache\_httpdconf.j2** template to **/etc/httpd/conf/httpd.conf**. The target file should be owned by **root** with **-r--r--r--** permissions. Restart Apache using the handler created previously.
- Copy the **apache\_indexhtml.j2** template to **/var/www/html/index.html**. The target file should be owned by **root** with **-r--r--r--** permissions.
- Start and enable the **httpd** and **firewalld** services.
- Open port 80/tcp on the firewall.

Package installation should always occur when this role is used, but the other tasks should only occur when the **apache\_enable** variable is set to **true**. The role should restart the Apache service when the configuration file is updated.

Modify **roles/myapache/tasks/main.yml** so that it contains the following:

```

tasks file for myapache

- name: install apache package
 yum:
```

```
 name: httpd
 state: latest

- name: install firewalld package
 yum:
 name: firewalld
 state: latest

- name: template out apache configuration file
 template:
 src: apache_httpdconf.j2
 dest: /etc/httpd/conf/httpd.conf
 owner: root
 group: root
 mode: 0444
 notify:
 - restart apache
 when: apache_enable

- name: template out apache index.html
 template:
 src: apache_indexhtml.j2
 dest: /var/www/html/index.html
 owner: root
 group: root
 mode: 0444
 when: apache_enable

- name: start and enable apache daemon
 service:
 name: httpd
 state: started
 enabled: true
 when: apache_enable

- name: start and enable firewalld daemon
 service:
 name: firewalld
 state: started
 enabled: true
 when: apache_enable

- name: open http firewall port
 firewalld:
 port: 80/tcp
 immediate: true
 permanent: true
 state: enabled
 when: apache_enable
```

13. Create the default variable values for the **myapache** role. The **apache\_enable** variable should have a default value of **false**.

Modify **roles/myapache/defaults/main.yml** so it contains the following:

```

defaults file for myapache

apache_enable: false
```

14. Create a playbook, called **apache.yml**, that runs on **serverb**. It should use the **myapache** role, but use the default value of the **apache\_enable** variable. Test the **myapache** role and confirm that it installs the packages, but does not deploy the web server.

- 14.1. Use a text editor to create the **apache.yml** playbook. It should look like the following:

```

- name: setup apache on serverb.lab.example.com
 hosts: serverb.lab.example.com
 roles:
 - myapache
```

- 14.2. Run the **apache.yml** playbook. Check the **ansible-playbook apache.yml** output to make sure it installs the needed packages, but skips the remaining tasks.

```
[student@workstation lab-roles]$ ansible-playbook apache.yml

PLAY [setup apache on serverb.lab.example.com] ****
TASK [Gathering Facts] ****
ok: [serverb.lab.example.com]

TASK [myapache : install apache package] ****
changed: [serverb.lab.example.com]

TASK [myapache : install firewalld package] ****
changed: [serverb.lab.example.com]

TASK [myapache : template out apache configuration file] ****
skipping: [serverb.lab.example.com]

TASK [myapache : template out apache index.html] ****
skipping: [serverb.lab.example.com]

TASK [myapache : start and enable apache daemon] ****
skipping: [serverb.lab.example.com]

TASK [myapache : open http firewall port] ****
skipping: [serverb.lab.example.com]

PLAY RECAP ****
serverb.lab.example.com : ok=3 changed=2 unreachable=0 failed=0
```

15. Modify the **apache.yml** playbook so that it uses the **myapache** role, setting the **apache\_enable** variable to **true**. Test the **myapache** role and confirm that it works properly.

- 15.1. Use a text editor to modify the **apache.yml** playbook. It should look like the following:

```

- name: setup apache on serverb.lab.example.com
 hosts: serverb.lab.example.com
 roles:
 - role: myapache
 apache_enable: true
```

15.2.Run the **apache.yml** playbook. Check the **ansible-playbook** output to make sure the tasks ran properly.

```
[student@workstation lab-roles]$ ansible-playbook apache.yml

PLAY [setup apache on serverb.lab.example.com] ****
TASK [Gathering Facts] ****
ok: [serverb.lab.example.com]

TASK [myapache : install apache package] ****
ok: [serverb.lab.example.com]

TASK [myapache : install firewalld package] ****
ok: [serverb.lab.example.com]

TASK [myapache : template out apache configuration file] ****
changed: [serverb.lab.example.com]

TASK [myapache : template out apache index.html] ****
changed: [serverb.lab.example.com]

TASK [myapache : start and enable apache daemon] ****
changed: [serverb.lab.example.com]

TASK [myapache : start and enable firewalld daemon] ****
changed: [serverb.lab.example.com]

TASK [myapache : open http firewall port] ****
changed: [serverb.lab.example.com]

RUNNING HANDLER [myapache : restart apache] ****
changed: [serverb.lab.example.com]

PLAY RECAP ****
serverb.lab.example.com : ok=9 changed=6 unreachable=0 failed=0
```

15.3.Use a web browser to confirm that **serverb** is serving web content.

```
[student@workstation lab-roles]$ curl -s http://serverb.lab.example.com
<!-- Ansible managed -->
<h2>Apache is running!</h2>
```

### Evaluation

Grade your work by running the **lab ansible-roles-lab grade** command from your **workstation** machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab ansible-roles-lab grade
```

### Cleanup

Run the **lab ansible-roles-lab cleanup** command to clean up the managed host.

```
[student@workstation ~]$ lab ansible-roles-lab cleanup
```

# Summary

In this chapter, you learned:

- Roles organize Ansible tasks in a way that allows reuse and sharing.
- Role variables should be defined in **defaults/main.yml** when they will be used as parameters, otherwise they should be defined in **vars/main.yml**.
- Role dependencies can be defined in the **dependencies** section of the **meta/main.yml** file of a role.
- Tasks that are to be applied before and after roles are included with the **pre\_tasks** and **post\_tasks** tasks in a playbook.
- Ansible roles are referenced in playbooks in the **roles** section.
- Default role variables can be overwritten when a role is used in a playbook.
- *Ansible Galaxy* [<https://galaxy.ansible.com>] is a public library of Ansible roles written by Ansible users.
- The **ansible-galaxy** command can search for, display information about, install, list, remove, or initialize roles.
- The **ansible-galaxy init --offline** command creates the well-defined directory structure that Ansible roles have.





## CHAPTER 8

# OPTIMIZING ANSIBLE

Overview	
<b>Goal</b>	Tune how Ansible executes plays and tasks using host patterns, delegation, and parallelism
<b>Objectives</b>	<ul style="list-style-type: none"><li>Specify managed hosts for plays and ad hoc commands using host patterns</li><li>Configure delegation in a playbook</li><li>Configure parallelism in Ansible</li></ul>
<b>Sections</b>	<ul style="list-style-type: none"><li>Selecting Hosts with Host Patterns (and Guided Exercise)</li><li>Configuring Delegation (and Guided Exercise)</li><li>Configuring Parallelism (and Guided Exercise)</li></ul>
<b>Lab</b>	<ul style="list-style-type: none"><li>Optimizing Ansible</li></ul>

# Selecting Hosts with Host Patterns

## Objectives

After completing this section, students should be able to:

- Explain how host patterns may be used to specify hosts from the inventory for plays or ad hoc commands.

## Referencing Inventory Hosts

*Host patterns* are used to specify the hosts which should be targeted by a play or ad hoc command. In its simplest form, the name of a managed host or a host group in the inventory is a host pattern that specifies that host or host group.

You've already been using host patterns in this course. In a play, the **hosts** directive specifies the managed hosts to run the play against. For an ad hoc command, the host pattern is provided as a command line argument to the **ansible** command.

Host patterns are important to understand. It is usually easier to control what hosts a play targets by carefully using host patterns and having appropriate inventory groups, instead of setting complex conditionals on the play's tasks.

### Managed Host

The most basic host pattern is the name for a single managed host listed in the inventory. This specifies that the host will be the only one in the inventory that will be acted upon by the **ansible** command.

The following example inventory will be used throughout this section to illustrate host patterns. The **ansible** command's **--list-hosts** option will be used to illustrate how some of these host patterns resolve.

The first example will specify a single managed host from the inventory.

```
[student@controlnode ~]$ cat myinventory
web.example.com
data.example.com

[lab]
labhost1.example.com
labhost2.example.com

[test]
test1.example.com
test2.example.com

[datacenter1]
labhost1.example.com
test1.example.com

[datacenter2]
labhost2.example.com
test2.example.com

[datacenter:children]
datacenter1
```

```
datacenter2
[new]
192.168.2.1
192.168.2.2

[student@controlnode ~]$ ansible web.example.com -i myinventory --list-hosts
hosts (1):
web.example.com
```

Remember that an IP address can be listed explicitly in the inventory instead of a host name. If it is, it can be used as a host pattern in the same way. If the IP address is not in the inventory, you can't use it to specify the host even if the IP address resolves to that host name.

The following example shows how a host pattern can be used to reference an IP address contained in an inventory.

```
[student@controlnode ~]$ ansible 192.168.2.1 -i myinventory --list-hosts
hosts (1):
192.168.2.1
```



## Note

One problem with referring to managed hosts by IP address in the inventory is that it can be hard to remember what IP address goes with which host for your plays and ad hoc commands. But you may find that you have to specify the host by IP address for connection purposes, because the host can't have a real host name in DNS for some reason.

It is possible to point an alias at a particular IP address in your inventory by setting the **ansible\_host** host variable. For example, you could have a host in your inventory named **dummy.example**, and then direct connections using that name to the IP address 192.168.2.1 by creating a **host\_vars/dummy.example** file containing the following host variable:

```
ansible_host: 192.168.2.1
```

## Groups

You've also already used inventory host groups as host patterns. When a group name is used as a host pattern, it specifies that Ansible will act on the hosts that are members of the group.

```
[student@controlnode ~]$ ansible lab -i myinventory --list-hosts
hosts (2):
labhost1.example.com
labhost2.example.com
```

Remember that there is a special group **all** that matches all managed hosts in the inventory.

```
[student@controlnode ~]$ ansible all -i myinventory --list-hosts
hosts (8):
web.example.com
data.example.com
```

```
labhost1.example.com
test1.example.com
labhost2.example.com
test2.example.com
192.168.2.1
192.168.2.2
```

There is also a special group **ungrouped** which matches all managed hosts in the inventory that are not members of any other group:

```
[student@controlnode ~]$ ansible ungrouped -i myinventory --list-hosts
hosts (2):
 web.example.com
 data.example.com
```

### Wildcards

Another method of accomplishing the same thing as the **all** host pattern is to use the asterisk (\*) wildcard character, which matches any string.

```
[student@controlnode ~]$ ansible '*' -i myinventory --list-hosts
hosts (8):
 labhost1.example.com
 test1.example.com
 labhost2.example.com
 test2.example.com
 web.example.com
 data.example.com
 192.168.2.1
 192.168.2.2
```



### Important

Some characters usable in host patterns also have meaning for the shell. This can be a problem when using host patterns to run ad hoc commands from the command line with **ansible**. In this case, it is a recommended practice to quote host patterns used on the command line to protect them from unwanted shell expansion.

Likewise, in an Ansible Playbook, you may need to put your host pattern in single quotes to ensure it's parsed correctly if you're using any special wildcards or list characters:

```
- name: Deploy to datacenter1 but not test1.example.com
 hosts: '!test1.example.com,development'
```

The asterisk character can also be used like file globbing to match any managed hosts or groups that contain a particular substring.

For example, the following wildcard host pattern matches all inventory names that end in **.example.com**:

```
[student@controlnode ~]$ ansible '*.example.com' -i myinventory --list-hosts
hosts (6):
 labhost1.example.com
 test1.example.com
```

```
labhost2.example.com
test2.example.com
web.example.com
data.example.com
```

The following example uses a wildcard host pattern to match names of hosts or host groups which start with **192.168.2.:**

```
[student@controlnode ~]$ ansible '192.168.2.*' -i myinventory --list-hosts
hosts (2):
 192.168.2.1
 192.168.2.2
```

The next example uses a wildcard host pattern to match names of hosts or host groups which begin with **datacenter**.

```
[student@controlnode ~]$ ansible 'datacenter*' -i myinventory --list-hosts
hosts (4):
 labhost1.example.com
 test1.example.com
 labhost2.example.com
 test2.example.com
```



## Important

The wildcard host patterns match all inventory names, hosts and host groups. They do not distinguish between names that are DNS names, IP addresses, or groups. This can lead to some unexpected matches if you forget this.

For example, given the example inventory, compare the results of specifying the **datacenter\*** host pattern from the preceding example with the results of the **data\*** host pattern:

```
[student@controlnode ~]$ ansible 'data*' -i myinventory --list-hosts
hosts (5):
 labhost1.example.com
 test1.example.com
 labhost2.example.com
 test2.example.com
 data.example.com
```

## Lists

Multiple entries in an inventory can be referenced using logical lists. A comma-separated list of host patterns matches all hosts that match any of those host patterns.

If you provide a comma-separated list of managed hosts, then all those managed hosts will be targeted:

```
[student@controlnode ~]$ ansible labhost1.example.com,test2.example.com,192.168.2.2 \
> -i myinventory --list-hosts
hosts (3):
 labhost1.example.com
 test2.example.com
```

```
192.168.2.2
```

If you provide a comma-separated list of groups, then all hosts in any of those groups will be targeted:

```
[student@controlnode ~]$ ansible lab,datacenter1 -i myinventory --list-hosts
hosts (3):
 labhost1.example.com
 labhost2.example.com
 test1.example.com
```

You can mix managed hosts, host groups, and wildcards as well:

```
[student@controlnode ~]$ ansible 'lab,data*,192.168.2.2' -i myinventory --list-hosts
hosts (6):
 labhost1.example.com
 labhost2.example.com
 test1.example.com
 test2.example.com
 data.example.com
 192.168.2.2
```



## Note

The colon character (:) can be used instead of a comma. However, comma is the preferred syntax, especially when working with IPv6 addresses as managed host names. You may see the colon syntax in older examples.

If an item in a list starts with an ampersand character (&), then hosts must match that item in order to match the host pattern. It operates similarly to a logical AND.

For example, based on our example inventory, the following host pattern will match machines in group **lab** only if they are also in group **datacenter1**:

```
[student@controlnode ~]$ ansible 'lab,&datacenter1' -i myinventory --list-hosts
hosts (1):
 labhost1.example.com
```

You could also specify that machines in group **datacenter1** match only if they're in group **lab** with the host patterns **&lab**, **datacenter1** or **datacenter1,&lab**.

You can exclude hosts which match a pattern from a list by using the exclamation point or "bang" character (!) in the front of the host pattern. This operates like a logical NOT.

This example, given our test inventory, matches all hosts defined in the **datacenter** group with the exception of **test2.example.com**:

```
[student@controlnode ~]$ ansible 'datacenter,!test2.example.com' -i myinventory \
> --list-hosts
hosts (3):
 labhost1.example.com
 test1.example.com
 labhost2.example.com
```

The pattern '**!test2.example.com, datacenter**' could have been used in the preceding example to get the same effect.

Finally, our last example shows the use of a host pattern that matches all hosts in our test inventory with the exception of the managed hosts in the **datacenter1** group.

```
[student@controlnode ~]$ ansible 'all,!datacenter1' -i myinventory --list-hosts
hosts (6):
web.example.com
data.example.com
labhost2.example.com
test2.example.com
192.168.2.1
192.168.2.2
```

## References

- Patterns – Ansible Documentation  
[http://docs.ansible.com/ansible/intro\\_patterns.html](http://docs.ansible.com/ansible/intro_patterns.html)
- Inventory – Ansible Documentation  
[http://docs.ansible.com/ansible/intro\\_inventory.html](http://docs.ansible.com/ansible/intro_inventory.html)

# Guided Exercise: Selecting Hosts with Host Patterns

In this exercise, you will explore how host patterns can be used to specify hosts from the inventory for plays or ad hoc commands. You will be provided with an example inventory to explore the host patterns.

## Outcomes

You should be able to use different host patterns to access various hosts in an inventory.

### Before you begin

Log in to **workstation** as **student** using **student** as the password. Run the **lab patterns setup** command.

```
[student@workstation ~]$ lab patterns setup
```

The setup script confirms that Ansible is installed on **workstation** and creates a directory structure for the lab environment.

### Steps

1. On **workstation**, change to the working directory for the exercise, **/home/student/patterns**.

```
[student@workstation ~]$ cd /home/student/patterns
[student@workstation patterns]$
```

- 1.1. List the contents of the directory.

```
[student@workstation patterns]$ ls
ansible.cfg inventory
```

- 1.2. Inspect the example inventory file. Notice how the inventory is organized. Explore which hosts are in the inventory, which domains are used, and which groups are in that inventory.

```
[student@workstation patterns]$ cat inventory
srv1.example.com
srv2.example.com
s1.lab.example.com
s2.lab.example.com

[web]
jupiter.lab.example.com
saturn.example.com

[db]
db1.example.com
db2.example.com
db3.example.com

[lb]
lb1.lab.example.com
```

```
lb2.lab.example.com

[boston]
db1.example.com
jupiter.lab.example.com
lb2.lab.example.com

[london]
db2.example.com
db3.example.com
file1.lab.example.com
lb1.lab.example.com

[dev]
web1.lab.example.com
db3.example.com

[stage]
file2.example.com
db2.example.com

[prod]
lb2.lab.example.com
db1.example.com
jupiter.lab.example.com

[function:children]
web
db
lb
city

[city:children]
boston
london
environments

[environments:children]
dev
stage
prod
new

[new]
172.25.252.23
172.25.252.44
172.25.252.32
```

2. Determine if the **db1.example.com** server is present in the inventory.

```
[student@workstation patterns]$ ansible db1.example.com -i inventory --list-hosts
hosts (1):
db1.example.com
```

3. Use an IP address host pattern to reference an IP address contained in the inventory.

```
[student@workstation patterns]$ ansible 172.25.252.44 -i inventory --list-hosts
hosts (1):
172.25.252.44
```

4. Use the **all** group to list all managed hosts in the inventory.

```
[student@workstation patterns]$ ansible all -i inventory --list-hosts
hosts (17):
 srv1.example.com
 srv2.example.com
 s1.lab.example.com
 s2.lab.example.com
 jupiter.lab.example.com
 saturn.example.com
 db1.example.com
 db2.example.com
 db3.example.com
 lb1.lab.example.com
 lb2.lab.example.com
 file1.lab.example.com
 web1.lab.example.com
 file2.example.com
 172.25.252.23
 172.25.252.44
 172.25.252.32
```

5. List all servers in the **london** group.

```
[student@workstation patterns]$ ansible london -i inventory --list-hosts
hosts (4):
 db2.example.com
 db3.example.com
 file1.lab.example.com
 lb1.lab.example.com
```

6. Access information about hosts in the **environments** nested group.

```
[student@workstation patterns]$ ansible environments -i inventory --list-hosts
hosts (10):
 web1.lab.example.com
 db3.example.com
 file2.example.com
 db2.example.com
 lb2.lab.example.com
 db1.example.com
 jupiter.lab.example.com
 172.25.252.23
 172.25.252.44
 172.25.252.32
```

7. List all hosts that do not belong to any group.

```
[student@workstation patterns]$ ansible ungrouped -i inventory --list-hosts
hosts (4):
 srv1.example.com
 srv2.example.com
 s1.lab.example.com
 s2.lab.example.com
```

8. Using the asterisk (\*) character, list all hosts that end in **.example.com**.

```
[student@workstation patterns]$ ansible '*.*example.com' -i inventory --list-hosts
hosts (14):
jupiter.lab.example.com
saturn.example.com
db1.example.com
db2.example.com
db3.example.com
lb1.lab.example.com
lb2.lab.example.com
file1.lab.example.com
web1.lab.example.com
file2.example.com
srv1.example.com
srv2.example.com
s1.lab.example.com
s2.lab.example.com
```

- As you can see in the output of the previous command, there are 14 hosts in the **\*.example.com** domain. Modify the search to ignore hosts in the **\*.lab.example.com** domain.

```
[student@workstation patterns]$ ansible '*.*example.com, !*.lab.example.com' \
> -i inventory --list-hosts
hosts (7):
saturn.example.com
db1.example.com
db2.example.com
db3.example.com
file2.example.com
srv1.example.com
srv2.example.com
```

- Without accessing the groups, list these three hosts: **lb1.lab.example.com**, **s1.lab.example.com** and **db1.example.com**.

```
[student@workstation patterns]$ ansible lb1.lab.example.com, \
> s1.lab.example.com, db1.example.com -i inventory --list-hosts
hosts (3):
lb1.lab.example.com
s1.lab.example.com
db1.example.com
```

- Use a wildcard host pattern to list hosts or host groups that start with a **172.25.** IP address.

```
[student@workstation patterns]$ ansible '172.25.*' -i inventory --list-hosts
hosts (3):
172.25.252.23
172.25.252.44
172.25.252.32
```

- List all hosts that start with the letter "s."

```
[student@workstation patterns]$ ansible 's*' -i inventory --list-hosts
hosts (7):
```

```
saturn.example.com
srv1.example.com
srv2.example.com
s1.lab.example.com
s2.lab.example.com
file2.example.com
db2.example.com
```

Notice the **file2.example.com** and **db2.example.com** hosts in the output of the previous command. They appear in the list because they are both members of a group called **stage**, which also begins with the letter "s."

13. Using a list and wildcard host pattern, list all hosts from the **prod** group, all hosts whose IP address begins with **172**, and all hosts that contain **lab** in their name.

```
[student@workstation patterns]$ ansible 'prod,172*,*lab*' -i inventory --list-hosts
hosts (11):
lb2.lab.example.com
db1.example.com
jupiter.lab.example.com
172.25.252.23
172.25.252.44
172.25.252.32
lb1.lab.example.com
file1.lab.example.com
web1.lab.example.com
s1.lab.example.com
s2.lab.example.com
```

14. Using a list, access all hosts that belong to both the **db** and **london** groups.

```
[student@workstation patterns]$ ansible 'db,&london' -i inventory --list-hosts
hosts (2):
db2.example.com
db3.example.com
```

# Configuring Delegation

## Objectives

After completing this section, students should be able to:

- Configure delegation in a playbook.

## Configuring delegation

In order to complete some configuration tasks, it may be necessary for actions to be taken on a different server than the one being configured. Some examples of this might include an action that requires waiting for the server to be restarted, adding a server to a load balancer or a monitoring server, or making changes to the DHCP or DNS database needed for the server being configured.

*Delegation* can help by performing necessary actions for tasks on hosts other than the managed host being targeted by the play in the inventory. Some scenarios that delegation can handle include:

- Delegating a task to the local machine
- Delegating a task to a host outside the play
- Delegating a task to a host that exists in the inventory
- Delegating a task to a host that does not exist in the inventory

### Delegating tasks to the local machine

When any action needs to be performed on the node running Ansible, it can be delegated to **localhost** by using **delegate\_to: localhost**.

Here is a sample playbook which, for each managed host, runs the command **ps** on the managed host and then on **localhost** by using the **delegate\_to** keyword. It displays the output from both **command** tasks using the **debug** module. Both tasks which run the **command** module set **changed\_when: false** because they don't change the state of either system when they run.

```

- name: delegate_to:localhost example
 hosts: dev
 tasks:
 - name: Get information on managed host processes
 command: ps
 register: remote_process
 changed_when: false

 - name: Display information on managed host processes
 debug:
 msg: "{{ remote_process.stdout }}"

 - name: Get information about localhost processes
 command: ps
 delegate_to: localhost
 register: local_process
 changed_when: false
```

```
- name: Display information on localhost processes
 debug:
 msg: "{{ local_process.stdout }}"
```

The **local\_action** keyword is a shorthand syntax which is sometimes used in place of **delegate\_to: localhost** on a per-task basis. After the colon, the module name should appear on the same line followed by its arguments using the obsolete key=value playbook syntax. Because it uses this older syntax, it can be harder to read or use **local\_action** instead of **delegate\_to: localhost**.

The previous playbook can be re-written using this shorthand syntax to delegate the task to the node running Ansible (**localhost**):

```

- name: local_action example
 hosts: dev
 tasks:
 - name: Get information on managed host processes
 command: ps
 register: remote_process
 changed_when: false

 - name: Display information on managed host processes
 debug:
 msg: "{{ remote_process.stdout }}"

 - name: Get information about localhost processes
 local_action: command ps
 register: local_process
 changed_when: false

 - name: Display information on localhost processes
 debug:
 msg: "{{ local_process.stdout }}"
```



## Important

Ansible 1.5 introduced the *implicit localhost* feature, which allows actions using "**hosts: localhost**" to succeed on the local machine when **localhost** is not defined in the inventory. In essence, with this feature the **localhost** managed host is implicitly defined by Ansible when no entry exists for it in the inventory.

The effect of this feature is evident in the resolution of the **localhost** host pattern even when no **localhost** entry is defined in the inventory.

```
[student@demo ~]$ cat inventory
[student@demo ~]$ ansible localhost --list-hosts
 hosts (1):
 localhost
```

It is important to note that **hosts: all** will *not* match the implicit localhost entry.

It is also important to note that the connection type for the implicit localhost definition is **local\_connection**. This is different from the default **ssh** connection type which would be used if **localhost** was explicitly defined in the inventory.

One side effect of this difference in connection type is that connections made using implicit localhost will ignore the **remote\_user** setting since there is no login process involved. If the **remote\_user** setting is defined to be different than the user that executes an Ansible command, administrators may experience different outcomes using implicit versus explicit **localhost** definitions.

For example, if privilege escalation with **sudo** is used, implicit localhost would be escalating privileges from the user account that executed the Ansible command, while explicit localhost would do so using the **remote\_user** account. If these two accounts were configured with different **sudo** privileges, the privilege escalation attempts may have different outcomes.

### Delegating task to a host outside the play

Ansible can be configured to run a task on a host other than the one that is part of the play with **delegate\_to**. The delegated module will still run once for every machine, but instead of running on the target machine, it will run on the host specified by **delegate\_to**. The facts available will be the ones applicable to the original host and not the host the task is delegated to. The task has the context of the original target host, but it gets executed on the host the task is delegated to.

The following example shows Ansible code that will delegate a task to an outside machine (in this case, **loadbalancer-host**). This example runs a command on the local balancer host to remove the managed hosts from the load balancer before deploying the latest version of the web stack. After that task is finished, a script is run to add the managed hosts back into the load balancer pool.

```
- hosts: webservers
 tasks:
 - name: Remove server from load balancer
 command: remove-from-lb {{ inventory_hostname }}
 delegate_to: loadbalancer-host
```

```
- name: deploy the latest version of web stack
 git:
 repo: git://foosball.example.org/path/to/repo.git
 dest: /srv/checkout

- name: Add server to load balancer pool
 command: add-to-lb {{ inventory_hostname }}
 delegate_to: loadbalancer-host
```

### Delegating a task to a host that exists in the inventory

When delegating to a host listed in the inventory, the inventory data will be used when creating the connection to the delegation target. This would include settings for **ansible\_connection**, **ansible\_host**, **ansible\_port**, **ansible\_user** and so on. Only the connection-related variables are used; the rest are read from the managed host originally targeted.

### Delegating a task to a host that does not exist in the inventory

When delegating a task to a host that does not exist in the inventory, Ansible will use the same connection type and details used for the managed host to connect to the delegating host. To adjust the connection details, use the **add\_host** module to create an ephemeral host in your inventory with connection data defined.

```
- name: test play
hosts: localhost
tasks:

- name: add delegation host
 add_host:
 name: demo
 ansible_host: 172.25.250.10
 ansible_user: devops

- name: echo Hello
 command: echo "Hello from {{ inventory_hostname }}"
 delegate_to: demo
 register: output

- debug:
 msg: "{{ output.stdout }}"
```

When the preceding playbook is executed, Ansible will use the connection details for the ephemeral host **demo** while executing the task **echo Hello** on the **delegate\_to** host. The **inventory\_hostname** will be read from the targeted managed host which in this case is **localhost**.

The following example shows the playbook execution. Note that the output contains the **add\_host** line and the variable expansion to **localhost**.

```
[student@demo ~]$ ansible-playbook test.yml -vvv
...
TASK [add delegation host] ****
task path: /home/student/ansible/dev-optimizing-ansible/inventory/test.yml:6
creating host via 'add_host': hostname=demo
changed: [localhost] => {"add_host": {"groups": [], "host_name": "demo", "host_vars": {"ansible_host": "172.25.250.10", "ansible_user": "devops"}, "changed": true, "invocation": {"module_args": {"ansible_host": "172.25.250.10", "ansible_user": "devops", "name": "demo"}, "module_name": "add_host"}}}
```

```
... Output omitted ...
TASK [echo Hello] ****
... Output omitted ...
changed: [localhost -> 172.25.250.10] => {"changed": true, "cmd": ["echo", "Hello from localhost"], "rc": 0}
```

### Task execution concurrency with delegation

Delegated tasks run for each managed host targeted. But Ansible tasks can run on multiple managed hosts in parallel. This can create issues with race conditions on the delegated host. This is particularly likely when using conditionals in the task, or when multiple concurrent tasks are run in parallel. This can also create a "thundering herd" problem where too many connections are being opened at once on the delegated host. SSH servers have a **MaxStartups** configuration option that can limit the number of concurrent connections allowed.

## Delegated Facts

Any facts gathered by a delegated task are assigned by default to the **delegate\_to** host, instead of the host which actually produced the facts. The following example shows a task file that will loop through a list of inventory servers to gather facts.

```
- hosts: app_servers
 tasks:
 - name: gather facts from app servers
 setup:
 delegate_to: "{{item}}"
 with_items: "{{groups['lb_servers']}}"

 - debug:
 var: ansible_eth0['ipv4']['address']
```

```
#inventory file
[app_servers]
demo.lab.example.com

[lb_servers]
workstation.lab.example.com
```

When the previous playbook is run, the output shows the gathered facts of **workstation.lab.example.com** as the task delegated to the host from the **lb\_servers** inventory group instead of **demo.lab.example.com**.

```
[student@demo ~]$ ansible-playbook delegatefacts.yml
... Output omitted ...
TASK [gather facts from app servers] ****
ok: [demo.lab.example.com -> workstation.lab.example.com] =>
 (item=workstation.lab.example.com)

TASK [debug] ****
ok: [demo.lab.example.com] => {
 "ansible_eth0['ipv4']['address']": "172.25.250.254"
}

PLAY RECAP ****
demo.lab.example.com : ok=3 changed=0 unreachable=0 failed=0
```

The **delegate\_facts** directive can be set to **True** to assign the gathered facts from the task to the delegated host instead of the current host.

```
- hosts: app_servers
 tasks:
 - name: gather facts from db servers
 setup:
 delegate_to: "{{item}}"
 delegate_facts: True
 with_items: "{{groups['lb_servers']}}"

 - debug:
 var: ansible_eth0['ipv4']['address']
```

On running the above playbook, the output now shows the facts gathered from **demo.lab.example.com** instead of the facts from the current managed host.

```
[student@demo ~]$ ansible-playbook delegatefacts.yml
... Output omitted ...
TASK [gather facts from db servers] ****
ok: [demo.lab.example.com -> workstation.lab.example.com] =>
 (item=workstation.lab.example.com)

TASK [debug] ****
ok: [demo.lab.example.com] => {
 "ansible_eth0['ipv4']['address']": "172.25.250.10"
}

PLAY RECAP ****
demo.lab.example.com : ok=3 changed=0 unreachable=0 failed=0
```

## References

- Delegation – Delegation, Rolling Updates, and Local Actions – Ansible Documentation  
[http://docs.ansible.com/ansible/playbooks\\_delegation.html#delegation](http://docs.ansible.com/ansible/playbooks_delegation.html#delegation)
- Delegated facts – Delegation, Rolling Updates, and Local Actions – Ansible Documentation  
[http://docs.ansible.com/ansible/playbooks\\_delegation.html#delegated-facts](http://docs.ansible.com/ansible/playbooks_delegation.html#delegated-facts)

# Guided Exercise: Configuring Delegation

In this exercise, you will configure the delegation of tasks in an Ansible playbook. The playbook will configure **serverc** as a proxy server and **servera** as an Apache web server. During the deployment of the website on **servera**, you will delegate the task of stopping the traffic coming to **servera** to **serverc** proxy server and later after deployment you will start the traffic coming to **servera** by delegating task to **serverc**.

## Outcomes

You should be able to:

- Configure the delegation of a task in a playbook.

## Before you begin

Log in to **workstation** as **student** using **student** as the password. Run the **lab configure-delegation setup** command.

```
[student@workstation ~]$ lab configure-delegation setup
```

The setup script confirms that Ansible is installed on **workstation** and creates a directory structure for the lab environment.

## Steps

- From **workstation**, as the **student** user, change to the **~/configure-delegation** directory.

```
[student@workstation ~]$ cd ~/configure-delegation
[student@workstation configure-delegation]$
```

- Create an inventory file named **hosts** under **~/configure-delegation/inventory**. The inventory file should have two groups defined: **webservers** and **proxyservers**. The **servera.lab.example.com** host should be part of the **webservers** group and **serverc.lab.example.com** should be part of the **proxyservers** group.

```
[webservers]
servera.lab.example.com

[proxyservers]
serverc.lab.example.com
```

- Move the **servera.lab.example.com-httpd.conf.j2**, template file that configures the virtual host to the **~/configure-delegation/templates** directory. Later you will use an Ansible variable (**inventory\_hostname**) to list the source of this file.

```
[student@workstation configure-delegation]$ mv servera.lab.example.com-httpd.conf.j2
~/configure-delegation/templates
```

- Move the **~/configure-delegation/serverc.lab.example.com-httpd.conf.j2** template file for configuring reverse proxy to the **~/configure-delegation/templates** directory.

```
[student@workstation configure-delegation]$ mv serverc.lab.example.com-httpd.conf.j2 ~/configure-delegation/templates
```

5. Create a template file, named **index.html.j2**, for the website to be hosted on **servera.lab.example.com** under the **templates** directory. The file should contain the following content:

```
The webroot is {{ ansible_fqdn }}.
```

6. Create a playbook named **site.yml** in the main project directory, **~/configure-delegation**. The playbook should define a play to install and configure *httpd*. The play should contain the following tasks:
- Install the *httpd* package and start and enable the service to **all** hosts defined in the inventory.
  - Configure firewall to accept incoming **http** traffic.
  - Copy the respective **httpd.conf** configuration file to the hosts serving as the web and proxy server. The resulting file should be named **myconfig.conf** under the **/etc/httpd/conf.d/myconfig.conf** directory.
- 6.1. Create a **site.yml** playbook under the lab project directory, **~/configure-delegation/**. Define a play inside the playbook that will execute the tasks on **all** hosts. Use **devops** for remote connection and enable privilege escalation to **root** using **sudo**.

```

- name: Install and configure httpd
 hosts: all
 remote_user: devops
 become: true
```

- 6.2. Continue editing the **site.yml** playbook. Define a task to install the *httpd* package and start and enable the **httpd** service on all hosts.

```
tasks:
 - name: Install httpd
 yum:
 name: httpd
 state: installed
 - name: Start and enable httpd
 service:
 name: httpd
 state: started
 enabled: yes
```

- 6.3. In the **site.yml** playbook, define a task to enable the firewall to allow web traffic on the managed hosts.

```
- name: Install firewalld
 yum:
 name: firewalld
```

```
state: installed
- name: Start and enable firewalld
 service:
 name: firewalld
 state: started
 enabled: yes
- name: Enable firewall
 firewalld:
 zone: public
 service: http
 permanent: true
 state: enabled
 immediate: true
```

- 6.4. Define a task in the **site.yml** playbook to copy the **serverc.lab.example.com-httpd.conf.j2** and **servera.lab.example.com-httpd.conf.j2** template files to the **/etc/httpd/conf.d/myconfig.conf** directory on their respective hosts. After copying the configuration file, use the **notify:** keyword to invoke the **restart httpd** handler defined in the next step.

```
- name: template server configs
 template:
 src: "templates/{{ inventory_hostname }}-httpd.conf.j2"
 dest: /etc/httpd/conf.d/myconfig.conf
 owner: root
 group: root
 mode: 0644
 notify:
 - restart httpd
```

- 6.5. In the **site.yml** playbook, define a handler to restart the **httpd** service when it is invoked.

```
handlers:
- name: restart httpd
 service:
 name: httpd
 state: restarted
```

- 6.6. Review the **site.yml** playbook. The file should contain the following:

```

- name: Install and configure httpd
 hosts: all
 remote_user: devops
 become: true
 tasks:
 - name: Install httpd
 yum:
 name: httpd
 state: installed
 - name: Start and enable httpd
 service:
 name: httpd
 state: started
 enabled: yes
 - name: Install firewalld
 yum:
```

```

 name: firewalld
 state: installed
 - name: Start and enable firewalld
 service:
 name: firewalld
 state: started
 enabled: yes
 - name: Enable firewall
 firewalld:
 zone: public
 service: http
 permanent: true
 state: enabled
 immediate: true
 - name: template server configs
 template:
 src: "templates/{{ inventory_hostname }}-httpd.conf.j2"
 dest: /etc/httpd/conf.d/myconfig.conf
 owner: root
 group: root
 mode: 0644
 notify:
 - restart httpd

 handlers:
 - name: restart httpd
 service:
 name: httpd
 state: restarted

```

7. Add another play to the **site.yml** playbook. The play should contain the following tasks:
- Stop the proxy server on **serverc.lab.example.com** using delegation.
  - Deploy the web page by copying the **index.html.j2** to the **/var/www/html/index.html** directory on **servera.lab.example.com**.
  - Start the proxy server on **serverc.lab.example.com** using delegation.

- 7.1. In the **site.yml** playbook, define another play that will have tasks to deploy a website that needs to be run on **servera.lab.example.com** of the **webservers** inventory group.

```

- name: Deploy web service and disable proxy server
 hosts: webservers
 remote_user: devops
 become: true

```

- 7.2. In the **site.yml** playbook, define a task to stop the incoming web traffic to **servera.lab.example.com** by stopping the proxy server running on **serverc.lab.example.com**.

Since the **hosts** keyword of the play points to **webservers** host group, use **delegate\_to** keyword to delegate this task to **serverc.lab.example.com** of the **proxyservers** host group.

```

tasks:
- name: Stop Apache proxy server
 service:

```

```
 name: httpd
 state: stopped
 delegate_to: "{{ item }}"
 with_items: "{{ groups['proxyservers'] }}"
```

- 7.3. In the **site.yml** playbook, define a task to copy the **index.html.j2** template present under **templates** directory to **/var/www/html/index.html** on **servera.lab.example.com**, part of the **webservers** group. Change the owner and group to **apache** and file permission to **0644**.

```
- name: Deploy webpages
 template:
 src: templates/index.html.j2
 dest: /var/www/html/index.html
 owner: apache
 group: apache
 mode: 0644
```

- 7.4. Add another task to the **site.yml** playbook that starts the proxy server by delegating the task to **serverc.lab.example.com**.

```
- name: Start Apache proxy server
 service:
 name: httpd
 state: started
 delegate_to: "{{ item }}"
 with_items: "{{ groups['proxyservers'] }}"
```

- 7.5. Review the **site.yml** playbook. The file should now contain the following additional content:

```
... Previous play content omitted ...

- name: Deploy web service and disable proxy server
 hosts: webservers
 remote_user: devops
 become: true
 tasks:
 - name: Stop Apache proxy server
 service:
 name: httpd
 state: stopped
 delegate_to: "{{ item }}"
 with_items: "{{ groups['proxyservers'] }}"
 - name: Deploy webpages
 template:
 src: templates/index.html.j2
 dest: /var/www/html/index.html
 owner: apache
 group: apache
 mode: 0644
 - name: Start Apache proxy server
 service:
 name: httpd
 state: started
 delegate_to: "{{ item }}"
 with_items: "{{ groups['proxyservers'] }}"
```

8. Check the syntax of the **site.yml** playbook. Resolve any syntax errors you find.

```
[student@workstation configure-delegation]$ ansible-playbook --syntax-check site.yml
playbook: site.yml
```

9. Execute the **site.yml** playbook. Watch for the delegation tasks in the command output.

```
[student@workstation configure-delegation]$ ansible-playbook site.yml

PLAY [Install httpd and configure] *****
... Output omitted ...

PLAY [Deploy web service and disable proxy server] *****

TASK [Gathering Facts] *****
ok: [servera.lab.example.com]

TASK [Stop Apache proxy server] *****
changed: [servera.lab.example.com -> serverc.lab.example.com] =>
(item=serverc.lab.example.com)

TASK [Deploy webpages] *****
changed: [servera.lab.example.com]

TASK [Start Apache proxy server] *****
changed: [servera.lab.example.com -> serverc.lab.example.com] =>
(item=serverc.lab.example.com)

PLAY RECAP *****
servera.lab.example.com : ok=12 changed=9 unreachable=0 failed=0
serverc.lab.example.com : ok=8 changed=3 unreachable=0 failed=0
```

10. Verify the website by browsing the <http://serverc.lab.example.com/external> link.

```
[student@workstation configure-delegation]$ curl http://serverc.lab.example.com/
external
The webroot is servera.lab.example.com.
```

### Cleanup

Run the **lab configure-delegation cleanup** command to cleanup after the lab.

```
[student@workstation ~]$ lab configure-delegation cleanup
```

# Configuring Parallelism

## Objectives

After completing this section, students should be able to:

- Configure parallelism in Ansible

## Configure parallelism in Ansible using forks

Ansible allows much more control over the execution of the playbook by running the tasks in parallel on all hosts. By default, Ansible only fork up to five times, so it will run a particular task on five different machines at once. This value is set in the Ansible configuration file **ansible.cfg**.

```
[student@demo ~]$ grep forks /etc/ansible/ansible.cfg
forks = 5
```

When there are a large number of managed hosts (more than five), the **forks** parameter can be changed to something more suitable for the environment. The default value can be either overridden in the configuration file by specifying a new value for the **forks** key, or the value can be changed using the **--forks** option for the **ansible-playbook** or **ansible** commands.

### Running tasks in parallel

For any specific play, you can use the **serial** keyword in a playbook to temporarily reduce the number of machines running in parallel from the fork count specified in the Ansible configuration file. The **serial** keyword is primarily used to control rolling updates.

### Rolling updates

If there is a website being deployed on 100 web servers, only 10 of them should be updated at the same time. The **serial** key can be set to 10 in the playbook to reduce the number of simultaneous deployments (assuming that the **fork** key was set to something higher). The **serial** keyword can also be specified as a percentage which will be applied to the total number of hosts in the play. If the number of hosts does not divide equally into the number of passes, the final pass will contain the modulus. Regardless of the percentage, the number of hosts per pass will always be 1 or greater.

```

- name: Limit the number of hosts this play runs on at the same time
 hosts: appservers
 serial: 2
```

Ansible, regardless of the number of forks set, only spins up the tasks based on the current number of hosts in a play.

## Asynchronous tasks

There are some system operations that take a while to complete. For example, when downloading a large file or rebooting a server, such tasks takes a long time to complete. Using parallelism and forks, Ansible starts the command quickly on the managed hosts, then polls the hosts for status until they are all finished.

To run an operation in parallel, use the **async** and **poll** keywords. The **async** keyword triggers Ansible to run the job in the background and can be checked later, and its value will be the maximum time that Ansible will wait for the command to complete. The value of **poll** indicates to Ansible how often to poll to check if the command has been completed. The default **poll** value is **10** seconds.

In the example, the **get\_url** module takes a long time to download a file and **async: 3600** instructs Ansible to wait for **3600** seconds to complete the task and **poll: 10** is the polling time in seconds to check if the download is complete.

```

```

```
- name: Long running task
 hosts: demoservers
 remote_user: devops
 tasks:
 - name: Download big file
 get_url:
 url: http://demo.example.com/bigfile.tar.gz
 async: 3600
 poll: 10
```

### Deferring asynchronous tasks

Long running operations or maintenance scripts can be carried out with other tasks, whereas checks for completion can be deferred until later using the **wait\_for** module. To configure Ansible to not wait for the job to complete, set the value of **poll** to **0** so that Ansible starts the command and instead of polling for its completion it moves to the next tasks.

```

```

```
- name: Restart and wait until the server is rebooted
 hosts: demoservers
 remote_user: devops
 tasks:
 - name: restart machine
 shell: sleep 2 && shutdown -r now "Ansible updates triggered"
 async: 1
 poll: 0
 become: true
 ignore_errors: true

 - name: waiting for server to come back
 wait_for:
 host: "{{ inventory_hostname }}"
 state: started
 delay: 30
 timeout: 300
 port: 22
 delegate_to: localhost
 become: false
```

For the running tasks that take an extremely long time to run, you can configure Ansible to wait for the job as long as it takes. To do this, set the value of **async** to **0**.

### Asynchronous task status

While an asynchronous task is running, you can also check its completion status by using Ansible **async\_status** module. The module requires the job or task identifier as its parameter.

```

```

```
Async status - fire-forget.yml
- name: Async status with fire and forget task
 hosts: demoservers
 remote_user: devops
 become: true
 tasks:

 - name: Download big file
 get_url:
 url: http://demo.example.com/bigfile.tar.gz
 async: 3600
 poll: 0
 register: download_sleeper

 - name: Wait for download to finish
 async_status:
 jid: "{{ download_sleeper.ansible_job_id }}"
 register: job_result
 until: job_result.finished
 retries: 30
```

The output of the playbook when executed:

```
[student@demo ~]$ ansible-playbook fire-forget.yml

PLAY [Async status with fire and forget task] ****
TASK [setup] ****
ok: [demo.example.com]

TASK [Download big file] ****
ok: [demo.example.com]

TASK [Wait for download to finish] ****
FAILED - RETRYING: TASK: Wait for download to fins (29 retries left). Result
was: {u'ansible_job_id': u'963772827414.1563', u'started': 1, u'changed': False,
u'finished': 0, u'results_file': u'/root/.ansible_async/963772827414.1563',
'invocation': {'module_name': u'async_status', 'module_args': {u'jid':
u'963772827414.1563', u'mode': u'status'}}}
... Output omitted ...
changed: [demo.example.com]

PLAY RECAP ****
demo.example.com : ok=3 changed=1 unreachable=0 failed=0
```



## References

**Rolling Update Batch Size – Delegation, Rolling Updates, and Local Actions – Ansible Documentation**

[http://docs.ansible.com/ansible/playbooks\\_delegation.html#rolling-update-batch-size](http://docs.ansible.com/ansible/playbooks_delegation.html#rolling-update-batch-size)

**Asynchronous Actions and Polling – Ansible Documentation**

[http://docs.ansible.com/ansible/playbooks\\_async.html](http://docs.ansible.com/ansible/playbooks_async.html)

**async\_status - Obtain status of asynchronous task – Ansible Documentation**

[http://docs.ansible.com/ansible/async\\_status\\_module.html](http://docs.ansible.com/ansible/async_status_module.html)

**Ansible Performance Tuning (For Fun and Profit)**

<https://www.ansible.com/blog/ansible-performance-tuning>

# Guided Exercise: Configuring Parallelism

In this exercise, you will run a playbook which uses a script to performs a long-running process on **servera.lab.example.com** using an asynchronous task. Instead of waiting for the task to get completed, you will check the status using the **async\_status** module.

## Outcomes

You should be able to:

- Configure parallelism using an asynchronous task in a playbook.

## Before you begin

Log in to **workstation** as **student** using **student** as the password.

On **workstation**, run the **lab configure-async setup** script. The setup script checks that Ansible is installed on **workstation**, creates a directory structure for the lab environment and an inventory file named **hosts**.

```
[student@workstation ~]$ lab configure-async setup
```

## Steps

- From **workstation**, as the **student** user, change to the directory **~/configure-async**.

```
[student@workstation ~]$ cd ~/configure-async
[student@workstation configure-async]$
```

- Create a script file named **longfiles.j2** under the **~/configure-async/templates** directory, with the following content.

```
#!/bin/bash
echo "emptying $2" > $2
for i in {00..30}; do
 echo "run $i, $1"
 echo "run $i for $1" >> $2
 sleep 1
done
```

- Create a playbook named **async.yml** under the lab's project directory. In the playbook, use the **webservers** inventory host group, the **devops** remote user, and for privilege escalation, use the **root** user and **sudo** method.

```
async.yml
- name: longfiles async playbook
 hosts: webservers
 remote_user: devops
 become: true
```

- In the playbook file **async.yml**, define a task to:

- Copy the **longfiles.j2** script under the **~/configure-async/templates** directory to the managed host under **/usr/local/bin/longfiles**.

- Change the file and group ownership to **root**, and change permission of the script to **0755**.

```
...output-omitted...
tasks:
 - name: template longfiles script
 template:
 src: templates/longfiles.j2
 dest: /usr/local/bin/longfiles
 owner: root
 group: root
 mode: 0755
```

5. Define a task and use the **async** keyword in the playbook file, **async.yml**, to:
  - Run the **longfiles** file copied previously under **/usr/local/bin/longfiles** with arguments **foo**, **bar**, and **baz** and their corresponding output files **/tmp/foo.file**, **/tmp/bar.file** and **/tmp/baz.file** respectively. For example: **/usr/local/bin/longfiles foo /tmp/foo.file**
  - Use the **async** keyword to wait the for the task for **110** second and set the value of **poll** to **0** so that Ansible starts the command and then runs in the background. Register a **script\_sleeper** variable to store the completion status of the command started.

```
...output-omitted...
- name: run longfiles script
 command: "/usr/local/bin/longfiles {{ item }} /tmp/{{ item }}.file"
 async: 110
 poll: 0
 with_items:
 - foo
 - bar
 - baz
 register: script_sleeper
```

6. In the playbook, **async.yml**, define a task to switch on the debug mode to see the value stored in the variable **script\_sleeper**.

```
...output-omitted...
- name: show script_sleeper value
 debug:
 var: script_sleeper
```

7. In the playbook, **async.yml**, define a task in the playbook that will keep checking the status of the **async** task:
  - Use the **async\_status** module to check the status of the **async** task triggered previously using the variable **script\_sleeper.result**.
  - Set the maximum retries to **30**.

```
...output-omitted...
- name: check status of longfiles script
 async_status: "jid={{ item.ansible_job_id }}"
 register: job_result
 until: job_result.finished
```

```
retries: 30
with_items: "{{ script_sleeper.results }}"
```

8. The completed **async.yml** playbook should have the following content:

```
async.yml
- name: longfiles async playbook
 hosts: webservers
 remote_user: devops
 become: true

 tasks:
 - name: template longfiles script
 template:
 src: templates/longfiles.j2
 dest: /usr/local/bin/longfiles
 owner: root
 group: root
 mode: 0755

 - name: run longfiles script
 command: "/usr/local/bin/longfiles {{ item }} /tmp/{{ item }}.file"
 async: 110
 poll: 0
 with_items:
 - foo
 - bar
 - baz
 register: script_sleeper

 - name: show script_sleeper value
 debug:
 var: script_sleeper

 - name: check status of longfiles script
 async_status: "jid={{ item.ansible_job_id }}"
 register: job_result
 until: job_result.finished
 retries: 30
 with_items: "{{ script_sleeper.results }}"
```

Check the syntax of the **async.yml** playbook. Correct any errors that you find.

```
[student@workstation configure-async]$ ansible-playbook --syntax-check async.yml
playbook: async.yml
```

9. Run the playbook **async.yml**.

Observe the job ids listed as **ansible\_job\_id** to each job running on **servera.lab.example.com**, that were run in parallel using the **async** keyword. The task *check status of longfiles script* retries to see if the started jobs are complete using the **async\_status** Ansible module.

```
[student@workstation configure-async]$ ansible-playbook async.yml
...output-omitted...
TASK [run longfiles script] ****
ok: [servera.lab.example.com] => (item=foo)
```

```
ok: [servera.lab.example.com] => (item=bar)
ok: [servera.lab.example.com] => (item=foo)
ok: [servera.lab.example.com] => (item=baz)

TASK [show script_sleeper value] *****
ok: [servera.lab.example.com] => {
 "script_sleeper": {
 "changed": false,
 "msg": "All items completed",
 "results": [
 {
 "_ansible_item_result": true,
 "_ansible_no_log": false,
 "_ansible_parsed": true,
 "ansible_job_id": "330645500410.4595",
 "changed": true,
 "finished": 0,
 "item": "foo",
 "results_file": "/root/.ansible_async/330645500410.4595",
 "started": 1
 },
 {
 "_ansible_item_result": true,
 "_ansible_no_log": false,
 "_ansible_parsed": true,
 "ansible_job_id": "408042741097.4686",
 "changed": true,
 "finished": 0,
 "item": "bar",
 "results_file": "/root/.ansible_async/408042741097.4686",
 "started": 1
 },
 {
 "_ansible_item_result": true,
 "_ansible_no_log": false,
 "_ansible_parsed": true,
 "ansible_job_id": "408778868074.4778",
 "changed": true,
 "finished": 0,
 "item": "baz",
 "results_file": "/root/.ansible_async/408778868074.4778",
 "started": 1
 }
]
 }
}

TASK [check status of longfiles script] *****
FAILED - RETRYING: TASK: check status of longfiles script (29 retries left).
...output-omitted...
FAILED - RETRYING: TASK: check status of longfiles script (25 retries left).
changed: [servera.lab.example.com] => (item={'_ansible_parsed': True, 'changed': True, '_ansible_no_log': False, u'ansible_job_id': u'330645500410.4595', u'started': 1, '_ansible_item_result': True, 'item': u'foo', u'finished': 0, u'results_file': u'/root/.ansib
le_async/330645500410.4595'})
changed: [servera.lab.example.com] => (item={'_ansible_parsed': True, 'changed': True, '_ansible_no_log': False, u'ansible_job_id': u'408042741097.4686', u'started': 1, '_ansible_item_result': True, 'item': u'bar', u'finished': 0, u'results_file': u'/root/.ansib
le_async/408042741097.4686'})
changed: [servera.lab.example.com] => (item={'_ansible_parsed': True, 'changed': True, '_ansible_no_log': False, u'ansible_job_id': u'408778868074.4778', u'started': 1, '_ansible_item_result': True, 'item': u'baz', u'finished': 0, u'results_file': u'/root/.ansib
```

```
le_async/408778868074.4778'})
PLAY RECAP ****
servera.lab.example.com : ok=5 changed=3 unreachable=0 failed=0
```

### Cleanup

Run the **lab configure-async cleanup** command to clean up the lab.

```
[student@workstation ~]$ lab configure-async cleanup
```

# Lab: Optimizing Ansible

In this lab, you will deploy an upgraded web page to add a new feature using the **serial** keyword for rolling updates, on two web servers, **serverb.lab.example.com** and **serverc.lab.example.com**, running behind a load balancer.

The HAProxy load balancer and the web servers are preconfigured on **serverd.lab.example.com**, **serverb.lab.example.com**, and **serverc.lab.example.com**, respectively.

After the upgrade of the web page, the web servers needs to be rebooted one at a time before adding them back to the load balancer pool without affecting the site availability using **delegation**.

## Outcomes

You should be able to write playbooks with tasks:

- To delegate tasks to other hosts.
- To asynchronously run jobs in parallel.
- To use rolling updates.

## Before you begin

Log in to **workstation** as **student**, using **student** as the password.

On **workstation**, run the **lab optimize-ansible-lab setup** script. It checks if Ansible is installed on **workstation** and creates a directory structure for the lab environment with an inventory file. The script preconfigures **serverb.lab.example.com** and **serverc.lab.example.com** as web servers and configures **serverd.lab.example.com** as the load balancer server using a round-robin algorithm. The script also creates a **templates** directory under the lab's working directory.

The inventory file, **/home/student/lab-optimizing-ansible/inventory/hosts**, points to **serverb.lab.example.com** and **serverc.lab.example.com** as hosts of **[webservers]** group and **serverd.lab.example.com** as part of the **[lbserver]** group.

```
[student@workstation ~]$ lab optimize-ansible-lab setup
```

## Steps

1. Log in to **workstation** as the **student** user and change to the **~/lab-optimizing-ansible** directory.

```
cd ~/lab-optimizing-ansible
[student@workstation lab-optimize-ansible]$
```

2. As the web servers are preconfigured as part of the lab setup, use **curl** to browse **http://serverd.lab.example.com**. Run the **curl** command twice to see the web content from the web servers running on **serverb** and **serverc** with **serverd** serving as the load balancer server.

- 
3. You will update the web site shown previously, by adding a new line to the web content. Modify the web page template, named **index-ver1.html.j2**, under the **templates** directory by adding the sentence: *A new feature added.*

```
<html>
<head><title>My Page</title></head>
<body>
<h1>
 Welcome to {{ inventory_hostname }}.
</h1>
<h2>A new feature added.</h2>
</body>
</html>
```

4. Create a playbook named **upgrade\_webserver.yml** under **~/lab-optimizing-ansible**. The playbook should use privilege escalation using the remote user **devops** and run on hosts from the **webservers** group.

The updates should be pushed to *one* server at a time.

5. Create a task in the **upgrade\_webserver.yml** playbook to remove the web server from the load balancer pool. Use the **haproxy** Ansible module to remove from the **haproxy** load balancer. The task needs to be delegated to server in the **[1bserver]** inventory group.

The **haproxy** module is used to disable a back end server from HAProxy using socket commands. To disable a back end server from the back end pool named **app**, socket path of **/var/lib/haproxy/stats**, and wait until the server reports a status of maintenance, use the following:

```
haproxy: state=disabled backend=app host={{ inventory_hostname }} socket=/var/lib/haproxy/stats wait=yes
```

6. Create a task in the **upgrade\_webserver.yml** playbook to copy the updated page template from the lab working directory, **templates/index-ver1.html.j2**, to the web server's document root, **/var/www/html/index.html**. Also register a variable, **pageupgrade**, which will be used later to invoke other tasks.
7. Create a task in the **upgrade\_webserver.yml** playbook to restart the web servers after **1** minute, using an *asynchronous* task that will not wait more than **1** second for it to complete. The task should not be polled for completion.

Use **ignore\_errors** as **true** and execute the task if the previously registered **pageupgrade** variable has changed.

8. Create a task in the **upgrade\_webserver.yml** playbook to wait for the web server to be rebooted. Use delegation to delegate the task to the local machine. The task should be executed when the variable, **pageupgrade**, has changed. Privilege escalation is not required for this task.

Use the **wait\_for** module to wait for the web server to be rebooted and set **host** to **inventory\_hostname**, **state** to **started**, **delay** to **80**, and **timeout** to **200**.

9. Create a task in the **upgrade\_webserver.yml** playbook to wait for the web server port to be started. As in the preconfigured web server, the **httpd** service is configured to start

at boot time. Set the **host** to **inventory\_hostname**, **port** to **80**, **state** to started, and **timeout** to **20**.

10. Create a final task in the **upgrade\_webserver.yml** playbook to add the web server to the load balancer pool after the upgrade of the web page. Use the **haproxy** Ansible module to add the server back to the **HAProxy** load balancer pool. The task needs to be delegated to all members of the **[lbservice]** inventory group.

The **haproxy** module is used to enable a back end server from **HAProxy** using socket commands. To enable a back end server from the back end pool named **app**, socket path of **/var/lib/haproxy/stats**, and to wait until the server reports healthy status, use the following:

```
haproxy: state=enabled backend=app host={{ inventory_hostname }} socket=/var/lib/haproxy/stats wait=yes
```

11. Review the **upgrade-webserver.yml** playbook file.
12. Check the syntax of the playbook **upgrade-webserver.yml**. In case of syntax errors resolve them before proceeding to the next step.

```
[student@workstation lab-optimizing-ansible]$ ansible-playbook --syntax-check
upgrade_webserver.yml

playbook: upgrade_webserver.yml
```

13. Run the playbook, **upgrade-webserver.yml**, to upgrade the server.

The restart task will take several minutes, so move on to the next step when it gets to that point in executing the playbook.



### Important

The playbook will wait at the **wait for server restart** task up to 2 minutes to allow the web server to reboot.

14. From **workstation.lab.example.com**, use **curl** to view the web link **http://serverd.lab.example.com**.  
  
Run the **curl** command several times while the playbook is executing to verify the webserver is still reachable, and that the host changes after the playbook moves on to reboot the other machine.
15. Wait until the remaining tasks from the playbook complete on both **serverb** and **serverc**.
16. Verify by browsing the website using the link **http://serverd.lab.example.com**. Rerun the **curl** command to see the updated pages from two different web servers, **serverb.lab.example.com** and **serverc.lab.example.com**.

---

#### Evaluation

From **workstation**, run the **lab optimize-ansible-lab** script with the **grade** argument to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab optimize-ansible-lab grade
```

#### Cleanup

Run the **lab optimize-ansible-lab cleanup** command to clean up after the lab.

```
[student@workstation ~]$ lab optimize-ansible-lab cleanup
```

## Solution

In this lab, you will deploy an upgraded web page to add a new feature using the **serial** keyword for rolling updates, on two web servers, **serverb.lab.example.com** and **serverc.lab.example.com**, running behind a load balancer.

The HAProxy load balancer and the web servers are preconfigured on **serverd.lab.example.com**, **serverb.lab.example.com**, and **serverc.lab.example.com**, respectively.

After the upgrade of the web page, the web servers needs to be rebooted one at a time before adding them back to the load balancer pool without affecting the site availability using **delegation**.

### Outcomes

You should be able to write playbooks with tasks:

- To delegate tasks to other hosts.
- To asynchronously run jobs in parallel.
- To use rolling updates.

### Before you begin

Log in to **workstation** as **student**, using **student** as the password.

On **workstation**, run the **lab optimize-ansible-lab setup** script. It checks if Ansible is installed on **workstation** and creates a directory structure for the lab environment with an inventory file. The script preconfigures **serverb.lab.example.com** and **serverc.lab.example.com** as web servers and configures **serverd.lab.example.com** as the load balancer server using a round-robin algorithm. The script also creates a **templates** directory under the lab's working directory.

The inventory file, **/home/student/lab-optimizing-ansible/inventory/hosts**, points to **serverb.lab.example.com** and **serverc.lab.example.com** as hosts of **[webservers]** group and **serverd.lab.example.com** as part of the **[lbserver]** group.

```
[student@workstation ~]$ lab optimize-ansible-lab setup
```

### Steps

1. Log in to **workstation** as the **student** user and change to the **~/lab-optimizing-ansible** directory.

```
[student@workstation ~]$ cd ~/lab-optimizing-ansible
[student@workstation lab-optimize-ansible]$
```

2. As the web servers are preconfigured as part of the lab setup, use **curl** to browse **http://serverd.lab.example.com**. Run the **curl** command twice to see the web content from the web servers running on **serverb** and **serverc** with **serverd** serving as the load balancer server.

```
[student@workstation lab-optimizing-ansible]$ curl http://serverd.lab.example.com
<html>
<head><title>My Page</title></head>
<body>
```

```

<h1>
 Welcome to serverc.lab.example.com.
</h1>
</body>
</html>
[student@workstation lab-optimizing-ansible]$ curl http://serverd.lab.example.com
<html>
<head><title>My Page</title></head>
<body>
<h1>
 Welcome to serverb.lab.example.com.
</h1>
</body>
</html>

```

3. You will update the web site shown previously, by adding a new line to the web content. Modify the web page template, named **index-ver1.html.j2**, under the **templates** directory by adding the sentence: *A new feature added.*

```

<html>
<head><title>My Page</title></head>
<body>
<h1>
 Welcome to {{ inventory_hostname }}.
</h1>
<h2>A new feature added.</h2>
</body>
</html>

```

4. Create a playbook named **upgrade\_webserver.yml** under **~/lab-optimizing-ansible**. The playbook should use privilege escalation using the remote user **devops** and run on hosts from the **webservers** group.

The updates should be pushed to *one* server at a time.

- 4.1. Create a playbook named **upgrade\_webserver.yml** under **~/lab-optimizing-ansible**. Use the hosts that are part of the **webservers** inventory group and use privilege escalation using remote user **devops**. Since the updates need to pushed to one server at a time, set the **serial** keyword with a value of **1**.

The contents of the **upgrade\_webserver.yml** file should be as follows:

```

- name: Upgrade Webservers
 hosts: webservers
 remote_user: devops
 become: yes
 serial: 1

```

5. Create a task in the **upgrade\_webserver.yml** playbook to remove the web server from the load balancer pool. Use the **haproxy** Ansible module to remove from the **haproxy** load balancer. The task needs to be delegated to server in the **[lserver]** inventory group.

The **haproxy** module is used to disable a back end server from HAProxy using socket commands. To disable a back end server from the back end pool named **app**, socket path of

**/var/lib/haproxy/stats**, and wait until the server reports a status of maintenance, use the following:

```
haproxy: state=disabled backend=app host={{ inventory_hostname }} socket=/var/lib/haproxy/stats wait=yes
```

- 5.1. Create a task in the **upgrade\_webserver.yml** playbook to remove the web server from the load balancer pool. The **haproxy** Ansible module can be used to remove the web server from the **haproxy** load balancer. The task needs to be delegated to the server from the **[lbserver]** inventory group. The contents of the **upgrade\_webserver.yml** file should appear as follows:

```
tasks:
 - name: disable the server in haproxy
 haproxy:
 state: disabled
 backend: app
 host: "{{ inventory_hostname }}"
 socket: /var/lib/haproxy/stats
 wait: yes
 delegate_to: "{{ item }}"
 with_items: "{{ groups.lbserver }}
```

6. Create a task in the **upgrade\_webserver.yml** playbook to copy the updated page template from the lab working directory, **templates/index-ver1.html.j2**, to the web server's document root, **/var/www/html/index.html**. Also register a variable, **pageupgrade**, which will be used later to invoke other tasks.

```
- name: upgrade the page
 template:
 src: "templates/index-ver1.html.j2"
 dest: "/var/www/html/index.html"
 register: pageupgrade
```

7. Create a task in the **upgrade\_webserver.yml** playbook to restart the web servers after **1** minute, using an *asynchronous* task that will not wait more than **1** second for it to complete. The task should not be polled for completion.

Use **ignore\_errors** as **true** and execute the task if the previously registered **pageupgrade** variable has changed.

- 7.1. Create a task in the **upgrade\_webserver.yml** playbook to reboot the web server by adding a task to the playbook. Use the **command** module to shut down the machine.

```
- name: restart machine
 command: shutdown -r +1 "Ansible updates triggered"
```

- 7.2. Continue editing the **restart machine** task in the **upgrade\_webserver.yml** playbook. Use the **async** keyword to wait for 1 second for the completion and set **poll** to **0** to disable polling. Set **ignore\_errors** to **true** and execute the task if the previously registered **pageupgrade** variable has changed. Add the lines in bold to the playbook.

```

- name: restart machine
 command: shutdown -r +1 "Ansible updates triggered"
 async: 1
 poll: 0
 ignore_errors: true
 when: pageupgrade.changed

```

8. Create a task in the **upgrade\_webserver.yml** playbook to wait for the web server to be rebooted. Use delegation to delegate the task to the local machine. The task should be executed when the variable, **pageupgrade**, has changed. Privilege escalation is not required for this task.

Use the **wait\_for** module to wait for the web server to be rebooted and set **host** to **inventory\_hostname**, **state** to **started**, **delay** to **80**, and **timeout** to **200**.

- 8.1. Create a task in the **upgrade\_webserver.yml** playbook. Delegate the task to **localhost**, and use the **wait\_for** module to wait for the server to restart. Set the **host** to the **inventory\_hostname** variable, **port** to **22**, **state** to **started**, **delay** to **80**, and **timeout** to **200**. The task should be executed when the variable, **pageupgrade**, has changed.

Privilege escalation is not required for this task. The task in the **upgrade\_webserver.yml** playbook should read as follows:

```

- name: wait for webserver to restart
 wait_for:
 host: "{{ inventory_hostname }}"
 port: 22
 state: started
 delay: 80
 timeout: 200
 become: False
 delegate_to: 127.0.0.1
 when: pageupgrade.changed

```

9. Create a task in the **upgrade\_webserver.yml** playbook to wait for the web server port to be started. As in the preconfigured web server, the **httpd** service is configured to start at boot time. Set the **host** to **inventory\_hostname**, **port** to **80**, **state** to **started**, and **timeout** to **20**.

```

- name: wait for webserver to come up
 wait_for:
 host: "{{ inventory_hostname }}"
 port: 80
 state: started
 timeout: 20

```

10. Create a final task in the **upgrade\_webserver.yml** playbook to add the web server to the load balancer pool after the upgrade of the web page. Use the **haproxy** Ansible module to add the server back to the **HAProxy** load balancer pool. The task needs to be delegated to all members of the **[lserver]** inventory group.

The **haproxy** module is used to enable a back end server from **HAProxy** using socket commands. To enable a back end server from the back end pool named **app**, socket path of **/var/lib/haproxy/stats**, and to wait until the server reports healthy status, use the following:

```
haproxy: state=enabled backend=app host={{ inventory_hostname }} socket=/var/lib/haproxy/stats wait=yes
```

Include the following content in the **upgrade\_webserver.yml** playbook:

```
- name: enable the server in haproxy
 haproxy:
 state: enabled
 backend: app
 host: "{{ inventory_hostname }}"
 socket: /var/lib/haproxy/stats
 wait: yes
 delegate_to: "{{ item }}"
 with_items: "{{ groups.lbserver }}"
```

11. Review the **upgrade-webserver.yml** playbook file.

The following should be the contents of **upgrade-webserver.yml**:

```

- name: Upgrade Webservers
 hosts: webservers
 remote_user: devops
 become: yes
 serial: 1

 tasks:
 - name: disable the server in haproxy
 haproxy:
 state: disabled
 backend: app
 host: "{{ inventory_hostname }}"
 socket: /var/lib/haproxy/stats
 wait: yes
 delegate_to: "{{ item }}"
 with_items: "{{ groups.lbserver }}"

 - name: upgrade the page
 template:
 src: "templates/index-ver1.html.j2"
 dest: "/var/www/html/index.html"
 register: pageupgrade

 - name: restart machine
 command: shutdown -r +1 "Ansible updates triggered"
 async: 1
 poll: 0
 ignore_errors: true
 when: pageupgrade.changed

 - name: wait for webserver to restart
 wait_for:
 host: "{{ inventory_hostname }}"
```

```

port: 22
state: started
delay: 80
timeout: 200
become: False
delegate_to: 127.0.0.1
when: pageupgrade.changed

- name: wait for webserver to come up
 wait_for:
 host: "{{ inventory_hostname }}"
 port: 80
 state: started
 timeout: 20

- name: enable the server in haproxy
 haproxy:
 state: enabled
 backend: app
 host: "{{ inventory_hostname }}"
 socket: /var/lib/haproxy/stats
 wait: yes
 delegate_to: "{{ item }}"
 with_items: "{{ groups.lbserver }}"

```

12. Check the syntax of the playbook **upgrade-webserver.yml**. In case of syntax errors resolve them before proceeding to the next step.

Check the syntax of the playbook **upgrade-webserver.yml**. In case of syntax errors, either resolve them or download the playbook from [http://materials.example.com/playbooks/upgrade\\_webserver.yml](http://materials.example.com/playbooks/upgrade_webserver.yml) before proceeding to the next step.

```
[student@workstation lab-optimizing-ansible]$ ansible-playbook --syntax-check
upgrade_webserver.yml

playbook: upgrade_webserver.yml
```

13. Run the playbook, **upgrade-webserver.yml**, to upgrade the server.

The restart task will take several minutes, so move on to the next step when it gets to that point in executing the playbook.



## Important

The playbook will wait at the **wait for server restart** task up to 2 minutes to allow the web server to reboot.

```
[student@workstation lab-optimizing-ansible]$ ansible-playbook upgrade_webserver.yml

PLAY [Upgrade Webservers] ****
TASK [Gathering Facts] ****
ok: [serverb.lab.example.com]

TASK [disable the server in haproxy] ****
changed: [serverb.lab.example.com -> serverd.lab.example.com] =>
 (item=serverd.lab.example.com)
```

```
TASK [upgrade the page] *****
changed: [serverb.lab.example.com]

TASK [restart machine] *****
[WARNING]: Module invocation had junk after the JSON data: Broadcast message from
root@serverb.lab.example.com (Mon 2017-06-26 11:33:03 EDT):
Ansible updates triggered The system is going down for reboot at Mon 2017-06-26
11:34:03 EDT!

changed: [serverb.lab.example.com]

TASK [wait for webserver to restart] *****
....output omitted....
```

14. From `workstation.lab.example.com`, use `curl` to view the web link `http://serverd.lab.example.com`.

Run the `curl` command several times while the playbook is executing to verify the webserver is still reachable, and that the host changes after the playbook moves on to reboot the other machine.

```
[student@workstation lab-optimizing-ansible]$ curl http://serverd.lab.example.com
<html>
<head><title>My Page</title></head>
<body>
<h1>
Welcome to serverc.lab.example.com.
</h1>
</body>
</html>
[student@workstation lab-optimizing-ansible]$ curl http://serverd.lab.example.com
<html>
<head><title>My Page</title></head>
<body>
<h1>
Welcome to serverb.lab.example.com.
</h1>
<h2>A new feature added.</h2>
</body>
</html>
```

15. Wait until the remaining tasks from the playbook complete on both `serverb` and `serverc`.

```
... Output omitted ...
TASK [wait for webserver to come up] *****
ok: [serverb.lab.example.com]

TASK [enable the server in haproxy] *****
changed: [serverb.lab.example.com -> serverd.lab.example.com] =>
(item=serverd.lab.example.com)

PLAY [Upgrade Webservers] *****
TASK [Gathering Facts] *****
ok: [serverc.lab.example.com]

TASK [disable the server in haproxy] *****
changed: [serverc.lab.example.com -> serverd.lab.example.com] =>
(item=serverd.lab.example.com)
```

```

TASK [upgrade the page] *****
changed: [serverc.lab.example.com]

TASK [restart machine] *****
[WARNING]: Module invocation had junk after the JSON data: Broadcast message from
root@serverc.lab.example.com (Mon 2017-06-26 11:35:03 EDT):
Ansible updates triggered The system is going down for reboot at Mon 2017-06-26
11:36:03 EDT!

changed: [serverc.lab.example.com]

TASK [wait for server to restart] *****
ok: [serverc.lab.example.com -> localhost]

TASK [wait for webserver to come up] *****
ok: [serverc.lab.example.com]

TASK [enable the server in haproxy] *****
changed: [serverc.lab.example.com -> serverd.lab.example.com] =>
(item=serverd.lab.example.com)

PLAY RECAP *****
serverb.lab.example.com : ok=7 changed=4 unreachable=0 failed=0
serverc.lab.example.com : ok=7 changed=4 unreachable=0 failed=0

```

16. Verify by browsing the website using the link **<http://serverd.lab.example.com>**. Rerun the **curl** command to see the updated pages from two different web servers, **serverb.lab.example.com** and **serverc.lab.example.com**.

```

[student@workstation lab-optimizing-ansible]$ curl http://serverd.lab.example.com
<html>
<head><title>My Page</title></head>
<body>
<h1>
Welcome to serverc.lab.example.com.
</h1>
<h2>A new feature added.</h2>
</body>
</html>
[student@workstation lab-optimizing-ansible]$ curl http://serverd.lab.example.com
<html>
<head><title>My Page</title></head>
<body>
<h1>
Welcome to serverb.lab.example.com.
</h1>
<h2>A new feature added.</h2>
</body>
</html>

```

#### Evaluation

From **workstation**, run the **lab optimize-ansible-lab** script with the **grade** argument to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab optimize-ansible-lab grade
```

#### Cleanup

Run the **lab optimize-ansible-lab cleanup** command to clean up after the lab.

```
[student@workstation ~]$ lab optimize-ansible-lab cleanup
```

# Summary

In this chapter, you learned:

- *Host patterns* are used to specify which managed hosts a play or ad hoc command should target
- Ansible can *delegate* tasks to run on a different host than the targeted managed host if actions need to be taken on a different server to configure the managed host
- Ansible normally attempts to run tasks in parallel on multiple hosts at the same time. The number of simultaneous connections can be controlled in the configuration file and in the playbook
- The **async** keyword triggers Ansible to run the job in the background, and the **poll** keyword controls how often Ansible will check to see if the task has finished





## CHAPTER 9

# IMPLEMENTING ANSIBLE VAULT

Overview	
<b>Goal</b>	Manage encryption with Ansible Vault.
<b>Objectives</b>	<ul style="list-style-type: none"><li>• Create, edit, rekey, encrypt, and decrypt files.</li><li>• Run a playbook with Ansible Vault.</li></ul>
<b>Sections</b>	<ul style="list-style-type: none"><li>• Configuring Ansible Vault (and Guided Exercise)</li><li>• Executing with Ansible Vault (and Guided Exercise)</li></ul>
<b>Lab</b>	<ul style="list-style-type: none"><li>• Implementing Ansible Vault</li></ul>

# Configuring Ansible Vault

## Objectives

After completing this section, students should be able to:

- Use Ansible Vault to create a new encrypted file or encrypt an existing file.
- Use Ansible Vault to view, edit, or change the password on an existing encrypted file.
- Remove encryption from a file that has been encrypted with Ansible Vault.

## Ansible Vault

Ansible may need access to sensitive data such as passwords or API keys in order to configure remote servers. Normally, this information might be stored as plain text in inventory variables or other Ansible files. But in that case, any user with access to the Ansible files or a version control system which stores the Ansible files would have access to this sensitive data. This poses an obvious security risk.

There are two primary ways to store this data more securely:

- Use **Ansible Vault**, which is included with Ansible and can encrypt and decrypt any structured data file used by Ansible.
- Use a third-party key management service to store the data in the cloud, such as **Vault** by HashiCorp, Amazon's **AWS Key Management Service**, or Microsoft Azure **Key Vault**.

In this part of the course, you will learn how to use Ansible Vault.

To use Ansible Vault, a command line tool called **ansible-vault** is used to create, edit, encrypt, decrypt, and view files. Ansible Vault can encrypt any structured data file used by Ansible. This might include inventory variables, included variable files in a playbook, variable files passed as an argument when executing the playbook, or variables defined in Ansible roles.



### Important

Ansible Vault does not implement its own cryptographic functions but uses an external Python toolkit. Files are protected with symmetric encryption using AES256 with a password as the secret key. Note that the way this is done has not been formally audited by a third party.

### Create an encrypted file

To create a new encrypted file use the command **ansible-vault create filename**. The command will prompt for the new vault password and open a file using the default editor. This is **vim**, but may be changed to **vi** in Ansible 2.1. A different editor may be used by setting and exporting the **\$EDITOR** variable. For example, to set the default editor to **nano**, **export EDITOR=nano**.

```
[student@demo ~]$ ansible-vault create secret.yml
New Vault password: redhat
```

```
Confirm New Vault password: redhat
```

Instead of entering the vault password through standard input, a vault password file can be used to store the vault password. This file will need to be carefully protected through file permissions and other means.

```
[student@demo ~]$ ansible-vault create --vault-password-file=vault-pass secret.yml
```

The cipher used to protect files is AES256 in recent versions of Ansible, but files encrypted with older versions may still use 128-bit AES.

#### Edit an existing encrypted file

To edit an existing encrypted file, Ansible Vault provides the command **ansible-vault edit filename**. This command will decrypt the file to a temporary file and allows you to edit the file. When saved, it copies the content and removes the temporary file.

```
[student@demo ~]$ ansible-vault edit secret.yml
Vault password: redhat
```



#### Note

The **edit** subcommand always rewrites the file, so it should only be used when making changes. This can have implications when the file is kept under version control. The **view** subcommand should always be used to see the file's contents without making changes.

#### Change the password for an encrypted file

The vault password can be changed using the command **ansible-vault rekey filename**. This command can rekey multiple data files at once. It will ask for the original password and the new password.

```
[student@demo ~]$ ansible-vault rekey secret.yml
Vault password: redhat
New Vault password: RedHat
Confirm New Vault password: RedHat
Rekey successful
```

When using vault password file, use the **--new-vault-password-file** option:

```
[student@demo ~]$ ansible-vault rekey \
> --new-vault-password-file=NEW_VAULT_PASSWORD_FILE secret.yml
```

#### Encrypting an existing file

To encrypt a file that already exists, use the command **ansible-vault encrypt filename**. This command can take the names of multiple files to be encrypted as arguments.

```
[student@demo ~]$ ansible-vault encrypt secret1.yml secret2.yml
New Vault password: redhat
Confirm New Vault password: redhat
Encryption successful
```

Use the **--output=OUTPUT\_FILE** option to save the encrypted file with a new name. At most one input file may be used with the **--output** option.

#### Viewing an encrypted file

Ansible Vault allows you to view the encrypted file using the command **ansible-vault view filename**, without opening it for editing.

```
[student@demo ~]$ ansible-vault view secret1.yml
Vault password: secret
less 458 (POSIX regular expressions)
Copyright (C) 1984-2012 Mark Nudelman

less comes with NO WARRANTY, to the extent permitted by law.
For information about the terms of redistribution,
see the file named README in the less distribution.
Homepage: http://www.greenwoodsoftware.com/less
my_secret: "yJJvPqhsiusmmPPZdnjndkdnyNDjdj782meUZcw"
```

#### Decrypting an existing file

An already existing encrypted file can be permanently decrypted by using the command **ansible-vault decrypt filename**. The **--output** option can be used when decrypting a single file to save the decrypted file under a different name.

```
[student@demo ~]$ ansible-vault decrypt secret1.yml --output=secret1-decrypted.yml
Vault password: redhat
Decryption successful
```

## References

**ansible-vault(1)** man page

Vault – Ansible Documentation  
[http://docs.ansible.com/ansible/playbooks\\_vault.html](http://docs.ansible.com/ansible/playbooks_vault.html)

# Guided Exercise: Configuring Ansible Vault

In this exercise, you will create a new encrypted file, edit the file, and change the password on an existing encrypted file. You will also learn how to encrypt and decrypt an existing file.

## Outcomes

You should be able to :

- Create a new encrypted file.
- Edit an encrypted file.
- View content of an encrypted file.
- Change the password of an encrypted file.
- Encrypt and decrypt an existing file.

## Before you begin

Log into **workstation** as **student** using **student** as a password.

On **workstation**, run the **lab configure-ansible-vault setup** script. The setup script checks that Ansible is installed on **workstation**, and creates a directory structure for the lab environment.

```
[student@workstation ~]$ lab configure-ansible-vault setup
```

## Steps

1. From **workstation**, as the **student** user, change to the directory **~/conf-ansible-vault**.

```
[student@workstation ~]$ cd ~/conf-ansible-vault
[student@workstation conf-ansible-vault]$
```

2. Create an encrypted file named **super-secret.yml** under **~/conf-ansible-vault**. Use **redhat** as the vault password.

- 2.1. Create an encrypted file named **super-secret.yml** under **~/conf-ansible-vault**. Enter **redhat** as the vault password when prompted, and confirm.

```
[student@workstation conf-ansible-vault]$ ansible-vault create super-secret.yml
New Vault password: redhat
Confirm New Vault password: redhat
```

- 2.2. Enter the following content into the file. Save and exit the file when you are finished.

```
This is encrypted.
```

3. Attempt to view the content of the encrypted file, **super-secret.yml**.

```
[student@workstation conf-ansible-vault]$ cat super-secret.yml
$ANSIBLE_VAULT;1.1;AES256
```

```
3035323263646262343861366639326339323861336333735626661646265376566653765633565
3663386561393538333864306136316265636632386535330a653764616133343630303633323831
33653136313933636633623431646634636661333762393764396135333236316338656338383933
3635646662316335370a363264366138333434626261363465636331333539323734643363326138
34626565353831666333653139323965376335633132313162613838613561396462323037313132
3264386531353862396233323963613139343635323532346538
```

Since the file, **super-secret.yml**, is an encrypted file, you cannot view the content in plain text. The default cipher used is *AES* (which is shared-secret based).

4. To view the content of the Ansible Vault encrypted file, use the command **ansible-vault view super-secret.yml**. When prompted, enter **redhat** as the password.

```
[student@workstation conf-ansible-vault]$ ansible-vault view super-secret.yml
Vault password: redhat
This is encrypted.
```

5. Now edit the encrypted file **super-secret.yml** to add some new content. Use **redhat** as the vault password.

- 5.1. Open the **super-secret.yml** encrypted file for editing.

```
[student@workstation conf-ansible-vault]$ ansible-vault edit super-secret.yml
Vault password: redhat
```

- 5.2. Add the following content to the file. Save and exit the file when you are finished.

```
This is also encrypted.
```

6. Verify by viewing the content of **super-secret.yml**, using **ansible-vault view super-secret.yml**. Use **redhat** as the password.

```
[student@workstation conf-ansible-vault]$ ansible-vault view super-secret.yml
Vault password: redhat
This is encrypted.
This is also encrypted.
```

7. Download the encrypted **passwd.yml** file from <http://materials.example.com/playbooks/passwd.yml>.

```
[student@workstation conf-ansible-vault]$ wget http://materials.example.com/
playbooks/passwd.yml
```

8. Change the vault password of the **passwd.yml** file, using the command **ansible-vault rekey passwd.yml**. The current password for the **passwd.yml** file is **redhat**. Change this password to **ansible**.

```
[student@workstation conf-ansible-vault]$ ansible-vault rekey passwd.yml
Vault password: redhat
New Vault password: ansible
Confirm New Vault password: ansible
Rekey successful
```

- 
9. Decrypt the encrypted file, **passwd.yml**, and save the file as **passwd-decrypted.yml**. Use the **ansible-vault decrypt** subcommand with the **--output** option. Enter **ansible** as the password.

```
[student@workstation conf-ansible-vault]$ ansible-vault decrypt passwd.yml --output=passwd-decrypted.yml
Vault password: ansible
Decryption successful
```

10. Verify the file **passwd-decrypted.yml** is decrypted by viewing its content using **cat**.

```
[student@workstation conf-ansible-vault]$ cat passwd-decrypted.yml
user_pw: 5pjsBJxAWUs6pRWD5ito/
```

11. Encrypt the existing file **passwd-decrypted.yml** and save the file as **passwd-encrypted.yml**. Use the **ansible-vault encrypt** subcommand with the **--output** option. Enter **redhat** as the password and confirm by re-entering the password.

```
[student@workstation conf-ansible-vault]$ ansible-vault encrypt passwd-decrypted.yml --output=passwd-encrypted.yml
New Vault password: redhat
Confirm New Vault password: redhat
Encryption successful
```

#### Evaluation

From **workstation**, run the **lab configure-ansible-vault** script with the **grade** argument, to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab configure-ansible-vault grade
```

# Executing with Ansible Vault

## Objectives

After completing this section, students should be able to:

- Run a playbook referencing files encrypted with Ansible Vault.

## Playbooks and Ansible Vault

In order to run a playbook that accesses files encrypted with Ansible Vault, the encryption password needs to be provided to the **ansible-playbook** command. If the command is run without doing this, it will return an error:

```
[student@demo ~]$ ansible-playbook site.yml
ERROR: A vault password must be specified to decrypt vars/api_key.yml
```

To provide the vault password interactively, use the **--ask-vault-pass** option.

```
[student@demo ~]$ ansible-playbook --ask-vault-pass site.yml
Vault password: redhat
```

Alternatively, a file that stores the encryption password in plain text can be used by specifying it with the **--vault-password-file** option. The password should be a string stored as a single line in the file. Since that file contains the sensitive plain text password, it is vital that it be protected through file permissions and other security measures.

```
[student@demo ~]$ ansible-playbook --vault-password-file=vault-pw-file site.yml
```

The default location of the password file can also be specified by using the **\$ANSIBLE\_VAULT\_PASSWORD\_FILE** environment variable.



### Important

All files protected by Ansible Vault that are used by a playbook must be encrypted using the same password.

### Recommended practices for variable file management

To simplify management, it makes sense to set up your Ansible project so that sensitive variables and all other variables are kept in separate files. The file or files containing the sensitive variables can then be protected with the **ansible-vault** command.

Remember that the preferred way to manage group variables and host variables is to create directories at the playbook level. The **group\_vars** directory normally contains variable files with names matching host groups to which they apply. The **host\_vars** directory normally contains variable files with names matching hostnames of managed hosts to which they apply.

However, instead of using files in **group\_vars** or **host\_vars**, you can use *directories* for each host group or managed host. Those directories can then contain multiple variable files, all of which are used by the host group or managed host. For example, in the following project

directory for **playbook.yml**, members of the **webservers** host group will use variables in the **group\_vars/webservers/vars** file, and **demo.example.com** will use the variables in both **host\_vars/demo.example.com/vars** and **host\_vars/demo.example.com/vault**:

```
.
└── ansible.cfg
└── group_vars
 └── webservers
 └── vars
└── host_vars
 └── demo.example.com
 ├── vars
 └── vault
└── inventory
└── playbook.yml
```

In this scenario, the advantage is that most variables for **demo.example.com** can be placed in **vars**, but sensitive variables can be placed in **vault**. Then the administrator can use **ansible-vault** to encrypt **vault**, while leaving **vars** as plain text.

There is nothing special about the file names being used in this example inside the **host\_vars/demo.example.com** directory. That directory could contain more files, some encrypted by Ansible Vault and some which are not.

Playbook variables (as opposed to inventory variables) can also be protected with Ansible Vault. Sensitive playbook variables can be placed in a separate file which is encrypted with Ansible Vault and which is included into the playbook through a **vars\_files** directive. This can be useful, since playbook variables take precedence over inventory variables.

#### Speeding up Vault operations

By default, Ansible uses functions from the *python-crypto* package to encrypt and decrypt vault files. If there are many encrypted files, decrypting them at start-up may cause a perceptible delay. To speed this up, install the *python-cryptography* package:

```
[student@demo ~]$ sudo yum install python-cryptography
```

The *python-cryptography* package provides a Python library which exposes cryptographic recipes and primitives. The default Ansible installation uses **PyCrypto** for these cryptographic operations.

## Demonstration: Executing with Ansible Vault

1. Log in to **workstation** as the **student** user. Jump to the **~/exec-ansible-vault** directory.

```
[student@workstation ~]$ cd ~/exec-ansible-vault
```

2. Create an encrypted file named **secret.yml** in **~/exec-ansible-vault/vars/** which will contain sensitive playbook variables. Provide a password of **redhat** for the vault and confirm it. This will open a new file in the default text editor, **vim**.

```
[student@workstation exec-ansible-vault]$ ansible-vault create vars/secret.yml
New Vault password: redhat
Confirm New Vault password: redhat
```

3. Once in the text editor, define an associative array variable called `newusers`. Each entry should have two keys: `name` for the username and `pw` for the password.

Define one user with a name of `demouser1` and a password of `redhat`. Define a second user with a name of `demouser2` and a password of `RedHat`.

```
newusers:
 - name: demouser1
 pw: redhat
 - name: demouser2
 pw: RedHat
```

Save the changes and exit the editor. This will create `vars/secret.yml`.

```
[student@workstation exec-ansible-vault]$ tree
.
├── ansible.cfg
├── createusers.yml
├── inventory
│ └── hosts
└── vars
 └── secret.yml

2 directories, 4 files
[student@workstation exec-ansible-vault]$ file vars/secret.yml
vars/secret.yml: ASCII text
```

4. Display the contents of the `create_users.yml` playbook. Note how it references `vars/secret.yml` as an external playbook variables file.

```

- name: create user accounts for all our servers
 hosts: devservers
 remote_user: devops
 become: yes
 vars_files:
 - vars/secret.yml
 tasks:
 - name: Creating users from secret.yml
 user:
 name: "{{ item.name }}"
 password: "{{ item.pw | password_hash('sha512') }}"
 with_items: "{{ newusers }}"
```

5. Use `ansible-playbook --syntax-check` to check the syntax of the `create_users.yml` playbook,

```
[student@workstation exec-ansible-vault]$ ansible-playbook --syntax-check \
> create_users.yml
ERROR! Decryption failed
```

It failed because it was unable to decrypt `vars/secret.yml` to check its syntax. Add the `--ask-vault-pass` option to prompt for the vault password while decrypting `vars/secret.yml`. In case of any syntax error, resolve before continuing further.

```
[student@workstation exec-ansible-vault]$ ansible-playbook --syntax-check \
> --ask-vault-pass create_users.yml
Vault password: redhat

playbook: create_users.yml
```

6. Create a password file, called **vault-pass**, to use for the playbook execution instead of asking for a password. Store the vault password **redhat** as plain text. Change the permission of the file to **0600**.

```
[student@workstation exec-ansible-vault]$ echo 'redhat' > vault-pass
[student@workstation exec-ansible-vault]$ chmod 0600 vault-pass
```

7. Execute the Ansible playbook, this time using the vault password file. This creates the **demouser1** and **demouser2** users on the managed hosts using the passwords stored as the **pw** fields in **secret.yml**.

```
[student@workstation exec-ansible-vault]$ ansible-playbook \
> --vault-password-file=vault-pass create_users.yml
```

8. Verify that both users (**demouser1** and **demouser2**) were created properly by the playbook. Connect to **servera.lab.example.com** via SSH as those users.

The **-o PreferredAuthentications=password** option must be used, because **servera** has been configured with SSH keys that permit authentication to the system without a password. In this case, we want to test out the password, so force SSH to ignore the SSH key.

- Log in to **servera.lab.example.com** as the **demouser1** user, using the password of **redhat**. Exit when you are finished.

```
[student@workstation exec-ansible-vault]$ ssh \
> -o PreferredAuthentications=password demouser1@servera.lab.example.com
demouser1@servera.lab.example.com's password: redhat
Warning: Permanently added 'servera.lab.example.com,172.25.250.10' (ECDSA) to the
list of known hosts.
[demouser1@servera ~]$ exit
```

- Log in to **servera.lab.example.com** as the **demouser2** user, using the password of **RedHat**. Exit when you are finished.

```
[student@workstation exec-ansible-vault]$ ssh \
> -o PreferredAuthentications=password demouser2@servera.lab.example.com
demouser2@servera.lab.example.com's password: RedHat
Last login: Fri Apr 8 10:30:58 2016 from workstation.lab.example.com
[demouser2@servera ~]$ exit
```



## References

**ansible-playbook(1)** and **ansible-vault(1)** man pages

Running a Playbook With Vault – Ansible Documentation

[http://docs.ansible.com/ansible/playbooks\\_vault.html#running-a-playbook-with-vault](http://docs.ansible.com/ansible/playbooks_vault.html#running-a-playbook-with-vault)

Variables and Vaults – Ansible Documentation

[http://docs.ansible.com/ansible/playbooks\\_best\\_practices.html#best-practices-for-variables-and-vaults](http://docs.ansible.com/ansible/playbooks_best_practices.html#best-practices-for-variables-and-vaults)

# Guided Exercise: Executing with Ansible Vault

In this exercise, you will use Ansible Vault to encrypt the file containing passwords on the local system and use the encrypted file in a playbook to create users on the **servera.lab.example.com** managed host.

## Outcomes

You should be able to:

- Use the variables defined in the encrypted file to execute a playbook.

## Before you begin

Log into **workstation** as **student** using **student** as a password.

On **workstation**, run the **lab execute-ansible-vault setup** script. This script ensures that Ansible is installed on **workstation** and creates a directory structure for the lab environment. This directory structure includes an inventory file that points to **servera.lab.example.com** as a managed host, which is part of the **devservers** group.

```
[student@workstation ~]$ lab execute-ansible-vault setup
```

## Steps

1. From **workstation**, as the **student** user, jump to the directory **~/exec-ansible-vault**.

```
[student@workstation ~]$ cd ~/exec-ansible-vault
```

2. Create an encrypted file named **secret.yml** in the **~/exec-ansible-vault/** directory. This file will define the password variables and store the passwords to be used in the playbook.

Use the associative array variable, **newusers**, to define users and passwords using the name and pw keys, respectively. Define the **ansibleuser1** user and its **redhat** password. Also define the **ansibleuser2** user and its **Re4H1T** password.

Set the vault password to **redhat**.

- 2.1. Create an encrypted file named **secret.yml** in **~/exec-ansible-vault/**. Provide a password of **redhat** for the vault and confirm it. This will open a file in the default editor **vim**.

```
[student@workstation exec-ansible-vault]$ ansible-vault create secret.yml
New Vault password: redhat
Confirm New Vault password: redhat
```

- 2.2. Add a associative array variable, named **newusers**, containing key value pair of user name and password as follows:

```
newusers:
 - name: ansibleuser1
 pw: redhat
 - name: ansibleuser2
```

```
pw: Re4H1T
```

Save the file.

3. Create a playbook which will use the variables defined in the **secret.yml** encrypted file. Name the playbook **create\_users.yml** and create it under the **~/exec-ansible-vault/** directory.

Configure the playbook to use the **devservers** host group, which was defined by the lab setup script in the inventory file. Run this playbook as the **devops** user on the remote managed host. Configure the playbook to create the **ansibleuser1** and **ansibleuser2** users.

The password stored as plain text in the variable, **pw**, should be converted into password hash using hashing filters **password\_hash** to get **SHA512** hashed password and passed as an argument to the **user** module. For example,

```
user:
 name: user1
 password: "{{ 'passwordsaresecret' | password_hash('sha512') }}"
```

The content of the **create\_users.yml** should be:

```

- name: create user accounts for all our servers
 hosts: devservers
 become: True
 remote_user: devops
 vars_files:
 - secret.yml
 tasks:
 - name: Creating users from secret.yml
 user:
 name: "{{ item.name }}"
 password: "{{ item.pw | password_hash('sha512') }}"
 with_items: "{{ newusers }}"
```

4. Check the syntax of the **create\_users.yml** playbook using **ansible-playbook --syntax-check**. Use the **--ask-vault-pass** option to prompt for the vault password which guards **secret.yml**. In case of syntax error, resolve before continuing further.

```
[student@workstation exec-ansible-vault]$ ansible-playbook --syntax-check --ask-vault-pass create_users.yml
Vault password: redhat
playbook: create_users.yml
```

5. Create a password file to use for the playbook execution instead of asking for a password. The file should be called **vault-pass** and it should store the **redhat** vault password as a plain text. Change the permission of the file to **0600**.

```
[student@workstation exec-ansible-vault]$ echo 'redhat' > vault-pass
[student@workstation exec-ansible-vault]$ chmod 0600 vault-pass
```

- 
6. Execute the Ansible playbook, using the vault password file to create the **ansibleuser1** and **ansibleuser2** users on a remote system using the passwords stored as variables in the **secret.yml** Ansible Vault encrypted file. Use the vault password file **vault-pass**.

```
[student@workstation exec-ansible-vault]$ ansible-playbook --vault-password-file=vault-pass create_users.yml

PLAY [create user accounts for all our servers] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]

TASK [Creating users from secret.yml] ****
changed: [servera.lab.example.com] => (item={'name': u'ansibleuser1', 'pw': u'redhat'})
changed: [servera.lab.example.com] => (item={'name': u'ansibleuser2', 'pw': u'Re4H1T'})

PLAY RECAP ****
servera.lab.example.com : ok=2 changed=1 unreachable=0 failed=0
```

7. Verify that both users were created properly by the playbook by connecting via SSH to **servera.lab.example.com**. Since we want to test the password, force SSH to skip the SSH key. Use the **-o PreferredAuthentications=password** option to SSH, because **servera** has been configured with an SSH key that will authenticate without a password.
- 7.1. Login to **servera.lab.example.com** as the **ansibleuser1** user, using the password of **redhat**, to verify the user was created properly by the playbook. Exit when you are finished.

```
[student@workstation exec-ansible-vault]$ ssh -o
PreferredAuthentications=password ansibleuser1@servera.lab.example.com
ansibleuser1@servera.lab.example.com's password: redhat
Warning: Permanently added 'servera.lab.example.com,172.25.250.10' (ECDSA) to
the list of known hosts.
[ansibleuser1@servera ~]$ exit
```

- 7.2. Login to **servera.lab.example.com** as the **ansibleuser2** user using the password of **Re4H1T** to verify the user was created properly by the playbook. Exit when you are finished.

```
[student@workstation exec-ansible-vault]$ ssh -o
PreferredAuthentications=password ansibleuser2@servera.lab.example.com
ansibleuser2@servera.lab.example.com's password: Re4H1T
[ansibleuser2@servera ~]$ exit
```

### Evaluation

From **workstation**, run the **lab execute-ansible-vault** script with the **grade** argument, to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab execute-ansible-vault grade
```

#### Cleanup

Run the **lab execute-ansible-vault cleanup** command to clean up the lab.

```
[student@workstation ~]$ lab execute-ansible-vault cleanup
```

# Lab: Implementing Ansible Vault

In this lab, you will encrypt and decrypt the YAML file containing variables for LUKS encryption which are sensitive. Use the encrypted file containing variables in a playbook to execute remote tasks on **serverb.lab.example.com** to create a LUKS encrypted partition on **/dev/vdb**. Edit the encrypted role variable file to add the path of the new 256-bit key file and add tasks to insert this key in an available key slot on the encrypted device, **/dev/vdb**, on **serverb.lab.example.com**.

## Outcomes

You should be able to:

- Encrypt a file.
- View an encrypted file.
- Run playbooks using an encrypted file.
- Change the password of an existing encrypted file.
- Edit and decrypt an existing encrypted file.

## Before you begin

Log into **workstation** as **student**, using **student** as a password. Run the **lab ansible-vault-lab setup** command.

```
[student@workstation ~]$ lab ansible-vault-lab setup
```

This command confirms that Ansible is installed on **workstation** and it creates a directory structure for the lab environment. It also creates the **~student/lab-ansible-vault/inventory/hosts** inventory file that includes **serverb.lab.example.com** as a managed host that is part of the **[prodservers]** host group.

## Steps

1. Log in as the **student** user on **workstation**. Change to the **~/lab-ansible-vault** project directory.
2. Use the **ansible-galaxy** command to create a role named **cryptodisk** and its directory structure.
3. Edit the role variable file to add the following variables:

Variable name	Variable value
luks_dev	/dev/vdb
luks_name	crypto
luks_pass	Re4H1TAns1BLe

```

vars file for cryptodisk
luks_dev: /dev/vdb
luks_name: crypto
```

```
luks_pass: Re4H1TAns1BLe
```

4. Encrypt the role variable file. Use **redhat** as the the vault password.
5. Create task to encrypt a block device as specified using the *luks\_dev* variable.

Download the task file from <http://materials.example.com/playbooks/encryptdisk-tasks.yml> and configure this file as a task for your playbook. These tasks will encrypt a block device using the **luks\_\*** variables defined in the role's variables file.

6. Create a playbook named **encrypt.yml** directly under the project directory that calls the **encryptdisk** role. Apply the role to the **prodservers** inventory host group. Do not forget to specify the remote user **devops** and enable privilege escalation.
7. Check the syntax of the playbook and ensure the roles are defined correctly. Use the **redhat** vault password when prompted.

If the playbook passed the syntax check, run the playbook to create an encrypted disk using the **/dev/vdb** block device on **serverb** and mount it under **/crypto**.

8. Verify the Ansible playbook created **/dev/vdb** as a LUKS disk partition and mounted it as **/crypto** on **serverb.lab.example.com**.
9. Download **keyfile-encrypted.j2** from <http://materials.example.com/playbooks/keyfile-encrypted.j2>. and rekey it. The original password for the file is **RedHat**. Change this password to **redhat**.
10. Permanently decrypt the Ansible Vault encrypted key file, **keyfile-encrypted.j2**, and name the new file **keyfile.j2**. This file will be added to an available key slot on the LUKS encrypted device on **serverb**. The vault password for this template file is **redhat**.

Store the decrypted **keyfile.j2** in the **encryptdisk** role's **templates** directory.

11. Edit the encrypted role variable file to add a new **luks\_key** variable which points to the decrypted **keyfile.j2** file created in the previous step. Use the vault password of **redhat**.
12. Add a default **addkey** variable in **roles/encryptdisk/defaults/main.yml** and set the value to **no**. This variable will be used as a conditional for adding a key file to the LUKS encrypted device.
13. Add a new task to the **encryptdisk** role to:
  - Copy the decrypted **keyfile.j2** key file from the role **templates** directory to **serverb.lab.example.com** as **/root/keyfile**. Set the owner to **root**, the group to **root**, and the mode to **0600**.
  - Use the **luks\_key** variable to substitute the path of the key file stored in the encrypted role's variables file. This task should be executed when the **addkey: yes** variable is defined when calling the **encryptdisk** role.
14. Add another task to the **encryptdisk** role to:
  - Use a command similar to the following example to add the key file to an available key slot on the encrypted disk on **serverb.lab.example.com**.

- Use the **luks\_pass** and **luks\_dev** variables to substitute the passphrase used earlier to encrypt the disk and the name of the device. This task should be invoked when the **addkey=yes** parameter is passed as an argument when running the playbook.

The following example shows how to add a key file to an available key slot on an encrypted LUKS disk. Replace *password* with the proper password for the device, *devicename* with the proper encrypted device name, and */path/to/keyfilename* with the proper path to the keyfile.

```
echo password | cryptsetup luksAddKey devicename /path/to/keyfilename
```

15. Edit the **~/lab-ansible-vault/encrypt.yml** playbook to specify the **addkey** variable to the **cryptdisk** role. Set the value of **addkey** to **yes**.
16. Check the syntax of the playbook and ensure roles are defined correctly. Use the **redhat** vault password when prompted.

If the playbook passed the syntax check, run the playbook to add the key file to the encrypted device on **serverb.lab.example.com**.

17. Verify the play worked correctly by accessing the LUKS encrypted file, using the key file added to the encrypted disk on **serverb.lab.example.com**. Use the following commands on **serverb** to sequentially verify the encrypted disk using the **cryptsetup** command:
  - 17.1. Dump the header information of a LUKS device using the **cryptsetup luksDump /dev/vdb** command.
  - 17.2. Unmount the mounted volume to close the encrypted device.
  - 17.3. Remove the existing **crypto** mapping and wipe the key from kernel memory.
  - 17.4. Open the **/dev/vdb** LUKS device and set up a **crypto** mapping after successful verification of the supplied **/root/keyfile** key file.
  - 17.5. Mount the filesystems found in **/etc/fstab**.
  - 17.6. List information about all available block devices.

Exit **serverb.lab.example.com** when finished.

#### Evaluation

Run the **lab ansible-vault-lab** command on **workstation**, with the *grade* argument, to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab ansible-vault-lab grade
```

#### Cleanup

Run the **lab ansible-vault-lab cleanup** command to clean up after the lab.

```
[student@workstation ~]$ lab ansible-vault-lab cleanup
```

## Solution

In this lab, you will encrypt and decrypt the YAML file containing variables for LUKS encryption which are sensitive. Use the encrypted file containing variables in a playbook to execute remote tasks on **serverb.lab.example.com** to create a LUKS encrypted partition on **/dev/vdb**. Edit the encrypted role variable file to add the path of the new 256-bit key file and add tasks to insert this key in an available key slot on the encrypted device, **/dev/vdb**, on **serverb.lab.example.com**.

### Outcomes

You should be able to:

- Encrypt a file.
- View an encrypted file.
- Run playbooks using an encrypted file.
- Change the password of an existing encrypted file.
- Edit and decrypt an existing encrypted file.

### Before you begin

Log into **workstation** as **student**, using **student** as a password. Run the **lab ansible-vault-lab setup** command.

```
[student@workstation ~]$ lab ansible-vault-lab setup
```

This command confirms that Ansible is installed on **workstation** and it creates a directory structure for the lab environment. It also creates the **~student/lab-ansible-vault/inventory/hosts** inventory file that includes **serverb.lab.example.com** as a managed host that is part of the **[prodservers]** host group.

### Steps

1. Log in as the **student** user on **workstation**. Change to the **~/lab-ansible-vault** project directory.

```
[student@workstation ~]$ cd ~/lab-ansible-vault
[student@workstation lab-ansible-vault]$
```

2. Use the **ansible-galaxy** command to create a role named **encryptdisk** and its directory structure.

```
[student@workstation lab-ansible-vault]$ ansible-galaxy init --offline -p roles/
 encryptdisk
 - encryptdisk was created successfully
```

3. Edit the role variable file to add the following variables:

Variable name	Variable value
luks_dev	/dev/vdb
luks_name	crypto

Variable name	Variable value
luks_pass	Re4H1TAns1BLe

Edit the **encrytdisk** role variable file, `~/lab-ansible-vault/roles/encrytdisk/vars/main.yml`, to add the following variables:

```

vars file for encrytdisk
luks_dev: /dev/vdb
luks_name: crypto
luks_pass: Re4H1TAns1BLe
```

- Encrypt the role variable file. Use **redhat** as the the vault password.

Use **ansible-vault** to encrypt the `roles/encrytdisk/vars/main.yml` role variable file.

```
[student@workstation lab-ansible-vault]$ ansible-vault encrypt roles/encrytdisk/
vars/main.yml
New Vault password: redhat
Confirm New Vault password: redhat
Encryption successful
```

Use **ansible-vault view** to display the encrypted file and confirm that the vault password works.

```
[student@workstation lab-ansible-vault]$ ansible-vault view roles/encrytdisk/vars/
main.yml
Vault password: redhat

vars file for encrytdisk
luks_dev: /dev/vdb
luks_name: crypto
luks_pass: Re4H1TAns1BLe
```

- Create task to encrypt a block device as specified using the `luks_dev` variable.

Download the task file from `http://materials.example.com/playbooks/encrytdisk-tasks.yml` and configure this file as a task for your playbook. These tasks will encrypt a block device using the `luks_*` variables defined in the role's variables file.

- Download the task file from `http://materials.example.com/playbooks/encrytdisk-tasks.yml`.

```
[student@workstation lab-ansible-vault]$ wget http://materials.example.com/
playbooks/encrytdisk-tasks.yml
```

- Move the task file to `roles/encrytdisk/tasks/main.yml`. This task will encrypt a block using the `luks_*` variables.

```
[student@workstation lab-ansible-vault]$ mv encrytdisk-tasks.yml roles/
encrytdisk/tasks/main.yml
```

6. Create a playbook named **encrypt.yml** directly under the project directory that calls the **cryptodisk** role. Apply the role to the **prodservers** inventory host group. Do not forget to specify the remote user **devops** and enable privilege escalation.

Create the **~/lab-ansible-vault/encrypt.yml** playbook with the following content:

```

- name: Encrypt disk on serverb using LUKS
 hosts: prodservers
 remote_user: devops
 become: yes
 roles:
 - cryptodisk
```

7. Check the syntax of the playbook and ensure the roles are defined correctly. Use the **redhat** vault password when prompted.

```
[student@workstation lab-ansible-vault]$ ansible-playbook --syntax-check --ask-vault-pass encrypt.yml
Vault password: redhat

playbook: encrypt.yml
```

If the playbook passed the syntax check, run the playbook to create an encrypted disk using the **/dev/vdb** block device on **serverb** and mount it under **/crypto**.

```
[student@workstation lab-ansible-vault]$ ansible-playbook --ask-vault-pass
 encrypt.yml
Vault password: redhat

PLAY [Encrypt disk on serverb using LUKS] ****
TASK [Gathering Facts] ****
ok: [serverb.lab.example.com]

TASK [cryptodisk : Check if device already unlocked.] ****
changed: [serverb.lab.example.com]

TASK [cryptodisk : Umount volumes] ****
skipping: [serverb.lab.example.com]

TASK [cryptodisk : Close disk... because of crypting/action requested] ****
skipping: [serverb.lab.example.com]

TASK [cryptodisk : Check if device already unlocked.] ****
changed: [serverb.lab.example.com]

TASK [cryptodisk : encrypt disk] ****
changed: [serverb.lab.example.com]

TASK [cryptodisk : open encrypted disk] ****
changed: [serverb.lab.example.com]

TASK [cryptodisk : Create directory /crypto] ****
changed: [serverb.lab.example.com]

TASK [cryptodisk : mkfs on /dev/vdb] ****
changed: [serverb.lab.example.com]
```

```

TASK [encryptdisk : create cryptab file] *****
changed: [serverb.lab.example.com]

TASK [encryptdisk : mount the /dev/mapper/crypto] *****
changed: [serverb.lab.example.com]

PLAY RECAP *****
serverb.lab.example.com : ok=9 changed=8 unreachable=0 failed=0

```

8. Verify the Ansible playbook created **/dev/vdb** as a LUKS disk partition and mounted it as **/crypto** on **serverb.lab.example.com**.

```

[student@workstation lab-ansible-vault]$ ansible prodservers -a 'lsblk'
serverb.lab.example.com | SUCCESS | rc=0 >>
NAME MAJ:MIN RM SIZE RO TYPE MOUNTPOINT
vda 253:0 0 40G 0 disk
└─vda1 253:1 0 40G 0 part /
vdb 253:16 0 1G 0 disk
└─crypto 252:0 0 1022M 0 crypt /crypto

```

9. Download **keyfile-encrypted.j2** from <http://materials.example.com/playbooks/keyfile-encrypted.j2>. and rekey it. The original password for the file is **RedHat**. Change this password to **redhat**.

```

[student@workstation lab-ansible-vault]$ wget http://materials.example.com/
playbooks/keyfile-encrypted.j2
... Output omitted ...
[student@workstation lab-ansible-vault]$ ansible-vault rekey keyfile-encrypted.j2
Vault password: RedHat
New Vault password: redhat
Confirm New Vault password: redhat
Rekey successful

```

10. Permanently decrypt the Ansible Vault encrypted key file, **keyfile-encrypted.j2**, and name the new file **keyfile.j2**. This file will be added to an available key slot on the LUKS encrypted device on **serverb**. The vault password for this template file is **redhat**.

Store the decrypted **keyfile.j2** in the **encryptdisk** role's **templates** directory.

```

[student@workstation lab-ansible-vault]$ ansible-vault decrypt keyfile-encrypted.j2
--output=roles/encryptdisk/templates/keyfile.j2
Vault password: redhat
Decryption successful

```

11. Edit the encrypted role variable file to add a new **luks\_key** variable which points to the decrypted **keyfile.j2** file created in the previous step. Use the vault password of **redhat**.

Edit the encrypted role variable file, **roles/encryptdisk/vars/main.yml**, to add a new variable **luks\_key: templates/keyfile.j2**.

```

[student@workstation lab-ansible-vault]$ ansible-vault edit roles/encryptdisk/vars/
main.yml
Vault password: redhat

```

```

vars file for encryptdisk
luks_dev: /dev/vdb
luks_name: crypto
luks_pass: Re4H1TAns1BLE
luks_key: templates/keyfile.j2
```

12. Add a default **addkey** variable in **roles/encryptdisk/defaults/main.yml** and set the value to **no**. This variable will be used as a conditional for adding a key file to the LUKS encrypted device.

```

defaults file for encryptdisk
addkey: no
```

13. Add a new task to the **encryptdisk** role to:
- Copy the decrypted **keyfile.j2** key file from the role **templates** directory to **serverb.lab.example.com** as **/root/keyfile**. Set the owner to **root**, the group to **root**, and the mode to **0600**.
  - Use the **luks\_key** variable to substitute the path of the key file stored in the encrypted role's variables file. This task should be executed when the **addkey: yes** variable is defined when calling the **encryptdisk** role.

Edit the role's task file, **roles/encryptdisk/tasks/main.yml**, to include the new task. The contents of the file should look like the following:

```

tasks file for Ansible Vault lab
... Output omitted ...
- name: copying the key file
 template:
 src: "{{ luks_key }}"
 dest: /root/keyfile
 owner: root
 group: root
 mode: 0600
 when: addkey
```

14. Add another task to the **encryptdisk** role to:
- Use a command similar to the following example to add the key file to an available key slot on the encrypted disk on **serverb.lab.example.com**.
  - Use the **luks\_pass** and **luks\_dev** variables to substitute the passphrase used earlier to encrypt the disk and the name of the device. This task should be invoked when the **addkey=yes** parameter is passed as an argument when running the playbook.

The following example shows how to add a key file to an available key slot on an encrypted LUKS disk. Replace **password** with the proper password for the device, **devicename** with the proper encrypted device name, and **/path/to/keyfilename** with the proper path to the keyfile.

```
echo password | cryptsetup luksAddKey devicename /path/to/keyfilename
```

Add the following lines to the **roles/encryptdisk/tasks/main.yml** task file:

```

tasks file for Ansible Vault lab
... Output omitted ...
- name: add new keyslot to encrypted disk
 shell: echo {{ luks_pass }} | cryptsetup luksAddKey {{ luks_dev }} /root/keyfile
 when: addkey
```

15. Edit the **~/lab-ansible-vault/encrypt.yml** playbook to specify the **addkey** variable to the **encryptdisk** role. Set the value of **addkey** to **yes**.

```

- name: Encrypt disk on serverb using LUKS
 hosts: prodservers
 remote_user: devops
 become: yes
 roles:
 - role: encryptdisk
 addkey: yes
```

16. Check the syntax of the playbook and ensure roles are defined correctly. Use the **redhat** vault password when prompted.

```
[student@workstation lab-ansible-vault]$ ansible-playbook --syntax-check --ask-vault-pass encrypt.yml
Vault password: redhat

playbook: encrypt.yml
```

If the playbook passed the syntax check, run the playbook to add the key file to the encrypted device on **serverb.lab.example.com**.

```
[student@workstation lab_ansiible_vault]$ ansible-playbook --ask-vault-pass
 encrypt.yml
Vault password: redhat
... Output omitted ...
TASK [encryptdisk : copying the key file] ****
changed: [serverb.lab.example.com]

TASK [encryptdisk : add new keyslot to encrypted disk] ****
changed: [serverb.lab.example.com]

PLAY RECAP ****
serverb.lab.example.com : ok=13 changed=10 unreachable=0 failed=0
```

17. Verify the play worked correctly by accessing the LUKS encrypted file, using the key file added to the encrypted disk on **serverb.lab.example.com**. Use the following commands on **serverb** to sequentially verify the encrypted disk using the **cryptsetup** command:

- 17.1. Dump the header information of a LUKS device using the **cryptsetup luksDump /dev/vdb** command.

```
[student@workstation lab-ansiible-vault]$ ssh root@serverb
```

```
[root@serverb ~]# cryptsetup luksDump /dev/vdb
... Output omitted ...
Key Slot 0: ENABLED
 Iterations: 319599
 Salt: 71 ba be b7 8a 1d db 4f c7 64 d0 58 3e 73 a8 48
 84 a3 4b 12 54 96 59 09 4d 3c 4e 24 d2 67 e7 a2
 Key material offset: 8
 AF stripes: 4000
Key Slot 1: ENABLED
 Iterations: 332899
 Salt: 55 30 60 ce 4c e5 a0 a6 49 fe e0 86 c0 56 da 2d
 56 0c 88 54 1c 07 27 27 18 b9 ec 22 99 e8 ab 2e
 Key material offset: 264
 AF stripes: 4000
... Output omitted ...
```

17.2. Unmount the mounted volume to close the encrypted device.

```
[root@serverb ~]# umount /crypto
```

17.3. Remove the existing **crypto** mapping and wipe the key from kernel memory.

```
[root@serverb ~]# cryptsetup close crypto
```

17.4. Open the **/dev/vdb** LUKS device and set up a **crypto** mapping after successful verification of the supplied **/root/keyfile** key file.

```
[root@serverb ~]# cryptsetup open /dev/vdb crypto -d /root/keyfile
```

17.5. Mount the filesystems found in **/etc/fstab**.

```
[root@serverb ~]# mount -a
```

17.6. List information about all available block devices.

```
[root@serverb ~]# lsblk
NAME MAJ:MIN RM SIZE RO TYPE MOUNTPOINT
vda 253:0 0 40G 0 disk
└─vda1 253:1 0 40G 0 part /
vdb 253:16 0 1G 0 disk
└─crypto 252:0 0 1022M 0 crypt /crypto
```

Exit **serverb.lab.example.com** when finished.

```
[root@serverb ~]# exit
[student@workstation lab-ansible-vault]$
```

### Evaluation

Run the **lab ansible-vault-lab** command on **workstation**, with the *grade* argument, to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab ansible-vault-lab grade
```

#### Cleanup

Run the **lab ansible-vault-lab cleanup** command to clean up after the lab.

```
[student@workstation ~]$ lab ansible-vault-lab cleanup
```

## Summary

In this chapter, you learned:

- *Ansible Vault* is one way to protect sensitive data like password hashes and private keys for deployment using Ansible playbooks
- Ansible Vault can use symmetric encryption (normally AES-256) to encrypt and decrypt any structured data file used by Ansible
- Ansible Vault can be used to create and encrypt a text file if it does not already exist or encrypt and decrypt existing files.
- All Vault files used by a playbook need to use the same password
- It is recommended that users keep most variables in a normal file and sensitive variables in a second file protected by Ansible Vault
- Ansible Vault operations can be run faster on Red Hat Enterprise Linux or CentOS by installing the *python-cryptography* package.



## CHAPTER 10

# TROUBLESHOOTING ANSIBLE

Overview	
<b>Goal</b>	Troubleshoot playbooks and managed hosts.
<b>Objectives</b>	<ul style="list-style-type: none"><li>• Troubleshoot playbooks</li><li>• Troubleshoot managed hosts</li></ul>
<b>Sections</b>	<ul style="list-style-type: none"><li>• Troubleshooting Playbooks (and Guided Exercise)</li><li>• Troubleshooting Ansible Managed Hosts (and Guided Exercise)</li></ul>
<b>Lab</b>	<ul style="list-style-type: none"><li>• Troubleshooting Ansible</li></ul>

# Troubleshooting Playbooks

## Objectives

After completing this section, students should be able to:

- Discuss common problems with playbooks
- Discuss tips to troubleshoot playbook issues
- Discuss recommended practices for playbook management

## Log Files in Ansible

By default, Ansible is not configured to log its output to any log file. It provides a built-in logging infrastructure that can be configured through the **log\_path** parameter in the **default** section of the **ansible.cfg** configuration file, or through the **\$ANSIBLE\_LOG\_PATH** environment variable. If any or both are configured, Ansible will store output from both the **ansible** and **ansible-playbook** commands in the log file configured either through the **ansible.cfg** configuration file, or the **\$ANSIBLE\_LOG\_PATH** environment variable.

If Ansible log files are to be kept in the default log file directory, **/var/log**, then the playbooks must be run as **root** or the permissions on **/var/log** must be opened up. More frequently log files are created in the local playbook directory.



### Note

Red Hat recommends that you configure **logrotate** to manage Ansible's log file.

## The Debug module

One of the modules available for Ansible, the *debug* module, provides a better insight into what is happening on the control node. This module can provide the value for a certain variable at playbook execution time. This feature is key to managing tasks that use variables to communicate with each other (for example, using the output of a task as the input to the following one). The following examples make use of the *msg* and *var* statements inside of the *debug* statement, to show the value at execution time of both the **ansible\_memfree\_mb** fact and the **output** variable.

```
- debug:
 msg: "The free memory for this system is {{ ansible_memfree_mb }}"
```

```
- debug:
 var: output
 verbosity: 2
```

## Managing errors

There are several issues than can occur during a playbook run, mainly related to the syntax of either the playbook or any of the templates it uses, or due to connectivity issues with the managed hosts (for example, an error in the host name of the managed host in the inventory file). Those errors are issued by the **ansible-playbook** command at execution time. The **--syntax-check** option checks the YAML syntax for the playbook. If a playbook has a high number of tasks, it may be useful to use either the **--step**, or the **--start-at-task** options in the **ansible-playbook** command. The **--step** option executes tasks interactively, asking if it should execute that task. The **--start-at-task** option starts the execution from a given task and avoid repeating all the previous tasks execution.

```
[student@demo ~]# ansible-playbook play.yml --step
```

```
[student@demo ~]# ansible-playbook play.yml --start-at-task="start httpd service"
```

```
[student@demo ~]# ansible-playbook play.yml --syntax-check
```

## Debugging with ansible-playbook

The output given by a playbook run with the **ansible-playbook** command is a good starting point to start troubleshooting issues related to hosts managed by Ansible. For example, a playbook is executed and the following output appears:

```
PLAY [playbook] ****
... Output omitted ...
TASK: [Install a service] ****
ok: [demoservera]
ok: [demoserverb]

PLAY RECAP ****
demoservera : ok=2 changed=0 unreachable=0 failed=0
demoserverb : ok=2 changed=0 unreachable=0 failed=0
```

The previous output shows a **PLAY** header with the name of the play to be executed, then one or more **TASK** headers are included. Each of these headers represent their associated *task* in the playbook, and it will be executed in all the managed hosts belonging to the group included in the playbook in the *hosts* parameter.

As each managed host executes the tasks, the host is displayed under the corresponding **TASK** header, along with the task state on that managed host, which can be set to **ok**, **fatal**, or **changed**. In the bottom of the playbook, at the **PLAY RECAP** section shows the number of tasks executed for each managed host as its output state.

The default output provided by the **ansible-playbook** command does not provide enough detail to troubleshoot issues that may appear on a managed host. The **ansible-playbook -v** command provides additional debugging information, with up to four total levels.

Verbosity configuration	
Option	Description
<b>-v</b>	The output data is displayed.

<b>Verbosity configuration</b>	
<b>-vv</b>	Both the output and input data are displayed.
<b>-vvv</b>	Includes information about connections to managed hosts.
<b>-vvvv</b>	Adds extra verbosity options to the connection plug-ins, including the users being used in the managed hosts to execute scripts, and what scripts have been executed.

## Recommended Practices for playbook management

Although the previously discussed tools can help to identify and fix issues in playbooks, when developing those playbooks it is important to keep in mind some recommended practices that can help ease the process to troubleshoot issues. Here are some of the recommended practices for playbook development.

- Always name tasks, providing a description in the **name** of the task's purpose. This **name** is displayed when the playbook is executed.
- Include comments to add additional inline documentation about tasks.
- Make use of vertical whitespace effectively. YAML syntax is mostly based on spaces, so avoid the usage of tabs in order to avoid errors.
- Try to keep the playbook as simple as possible. Only use the features that you need.

## References

- log\_path (Configuration file – Ansible Documentation)  
[http://docs.ansible.com/ansible/intro\\_configuration.html#log-path](http://docs.ansible.com/ansible/intro_configuration.html#log-path)
- debug – Print statements during execution – Ansible Documentation  
[http://docs.ansible.com/ansible/debug\\_module.html](http://docs.ansible.com/ansible/debug_module.html)
- Best Practices – Ansible Documentation  
[http://docs.ansible.com/ansible/playbooks\\_best\\_practices.html](http://docs.ansible.com/ansible/playbooks_best_practices.html)

# Guided Exercise: Troubleshooting Playbooks

In this exercise, a playbook has errors that need to be corrected. The project structure from previous units is going to be reused. The playbook for this exercise is the **samba.yml** playbook that configures a Samba service on **servera.lab.example.com**.

## Outcomes

You should be able to:

- Troubleshoot playbooks.

## Before you begin

Log in to **workstation** as **student** using **student** as the password.

On **workstation**, run the **lab troubleshoot-playbooks setup** script. It checks if Ansible is installed on **workstation**. It also creates the **/home/student/troubleshooting/** directory, and downloads to this directory the **inventory**, **samba.yml**, and **samba.conf.j2** files from **http://materials.example.com/troubleshooting/**.

```
[student@workstation ~]$ lab troubleshoot-playbooks setup
```

## Steps

1. On workstation, change to the **/home/student/troubleshooting/** directory.

```
[student@workstation ~]$ cd ~/troubleshooting/
```

2. Create a file named **ansible.cfg** in the current directory as follows, configuring the **log\_path** parameter for Ansible to start logging to the **/home/student/troubleshooting/ansible.log** file, and the **inventory** parameter to use the **/home/student/troubleshooting/inventory** file deployed by the lab script. When you are finished, **ansible.cfg** should have the following contents:

```
[defaults]
log_path = /home/student/troubleshooting/ansible.log
inventory = /home/student/troubleshooting/inventory
```

3. Run the playbook. This playbook sets up a Samba server if everything is correct. The run will fail due to missing double quotes on the **random\_var** variable definition. The error message is very indicative of the issue with the run. Notice the variable **random\_var** is assigned a value that contains a colon and is not quoted.

```
[student@workstation troubleshooting]$ ansible-playbook samba.yml
ERROR! Syntax Error while loading YAML.
```

The error appears to have been in '/home/student/troubleshooting/samba.yml': line 8, column 30, but may be elsewhere in the file depending on the exact syntax problem.

The offending line appears to be:

```
install_state: installed
random_var: This is colon: test
```

^ here

4. Check that the error has been properly logged to the **/home/student/troubleshooting/ansible.log** file.

```
[student@workstation troubleshooting]$ tail ansible.log
The error appears to have been in '/home/student/troubleshooting/samba.yml': line 8,
column 30, but may
be elsewhere in the file depending on the exact syntax problem.
```

The offending line appears to be:

```
install_state: installed
random_var: This is colon: test
 ^ here
```

5. Edit the playbook and correct the error, and add quotes to the entire value being assigned to **random\_var**. The corrected edition of **samba.yml** should contain the following content:

```
... Output omitted ...
vars:
 install_state: installed
 random_var: "This is colon: test"
... Output omitted ...
```

6. Run the playbook using the **--syntax-check** option. An error is issued due to the extra space in the indentation on the last task, **deliver samba config**.

```
[student@workstation troubleshooting]$ ansible-playbook samba.yml --syntax-check
ERROR! Syntax Error while loading YAML.
```

The error appears to have been in '/home/student/troubleshooting/samba.yml': line 43, column 4, but may  
be elsewhere in the file depending on the exact syntax problem.

The offending line appears to be:

```
- name: deliver samba config
 ^ here
```

7. Edit the playbook and remove the extra space for all lines in that task. The corrected playbook content should look like the following:

```
... Output omitted ...
- name: configure firewall for samba
 firewalld:
 state: enabled
 permanent: true
 immediate: true
 service: samba

- name: deliver samba config
 template:
 src: templates/samba.conf.j2
```

```
dest: /etc/samba/smb.conf
owner: root
group: root
mode: 0644
```

8. Run the playbook using the **--syntax-check** option. An error is issued due to the **install\_state** variable being used as a parameter in the **install samba** task. It is not quoted.

```
[student@workstation troubleshooting]$ ansible-playbook samba.yml --syntax-check
ERROR! Syntax Error while loading YAML.
```

The error appears to have been in '/home/student/troubleshooting/samba.yml': line 14, column 15, but may be elsewhere in the file depending on the exact syntax problem.

The offending line appears to be:

```
name: samba
state: {{ install_state }}
^ here
```

We could be wrong, but this one looks like it might be an issue with missing quotes. Always quote template expression brackets when they start a value. For instance:

```
with_items:
- {{ foo }}
```

Should be written as:

```
with_items:
- "{{ foo }}"
```

9. Edit the playbook and correct the **install samba** task. The reference to the **install\_state** variable should be in quotes. The resulting file content should look like the following:

```
... Output omitted ...
tasks:
- name: install samba
 yum:
 name: samba
 state: "{{ install_state }}"
... Output omitted ...
```

10. Run the playbook using the **--syntax-check** option. It should not show any additional syntax errors.

```
[student@workstation troubleshooting]$ ansible-playbook samba.yml --syntax-check
playbook: samba.yml
```

11. Run the playbook. An error, related to SSH, will be issued.

```
[student@workstation troubleshooting]$ ansible-playbook samba.yml
```

```
PLAY [Install a samba server] ****
TASK [Gathering Facts] ****
fatal: [servera.lab.exammples.com]: UNREACHABLE! => {"changed": false,
 "msg": "Failed to connect to the host via ssh: ssh: Could not resolve hostname
 servera.lab.exammples.com: Name or service not known\r\n", "unreachable": true}
 to retry, use: --limit @/home/student/troubleshooting/samba.retry

PLAY RECAP ****
servera.lab.exammples.com : ok=0 changed=0 unreachable=1 failed=0
```

12. Ensure the managed host **servera.lab.example.com** is running, using the **ping** command.

```
[student@workstation troubleshooting]$ ping -c3 servera.lab.example.com
PING servera.lab.example.com (172.25.250.10) 56(84) bytes of data.
64 bytes from servera.lab.example.com (172.25.250.10): icmp_seq=1 ttl=64 time=0.247
ms
64 bytes from servera.lab.example.com (172.25.250.10): icmp_seq=2 ttl=64 time=0.329
ms
64 bytes from servera.lab.example.com (172.25.250.10): icmp_seq=3 ttl=64 time=0.320
ms

--- servera.lab.example.com ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1999ms
rtt min/avg/max/mdev = 0.247/0.298/0.329/0.041 ms
```

13. Ensure that you can connect to the managed host **servera.lab.example.com** as the **devops** user using SSH, and that the correct SSH keys are in place. Log out again when you have finished.

```
[student@workstation troubleshooting]$ ssh devops@servera.lab.example.com
Warning: Permanently added 'servera.lab.example.com,172.25.250.10' (ECDSA) to the
list of known hosts.
... Output omitted ...
[devops@servera ~]$ exit
Connection to servera.lab.example.com closed.
```

14. Rerun the playbook with **-vvvv** to get more information about the run. An error is issued because the **servera.lab.example.com** managed host is not reachable.

```
[student@workstation troubleshooting]$ ansible-playbook -vvvv samba.yml
Using /etc/ansible/ansible.cfg as config file
Loaded callback default of type stdout, v2.0
1 plays in samba.yml

PLAY [Install a samba server] ****
TASK [Gathering Facts] ****
<servera.lab.exampple.com> ESTABLISH SSH CONNECTION FOR USER: devops
... Output omitted ...
fatal: [servera.lab.exampple.com]: UNREACHABLE! => { "changed": false, "msg":
 "Failed to connect to the host via ssh: OpenSSH_6.6.1, OpenSSL 1.0.1e-fips
 11 Feb 2013\r\ndebug1: Reading configuration data /home/student/.ssh/config\r\n
\r\ndebug1: /home/student/.ssh/config line 1: Applying options for *\r\ndebug1:
 Reading configuration data /etc/ssh/ssh_config\r\ndebug1: /etc/ssh/ssh_config line
 56: Applying options for *\r\ndebug1: auto-mux: Trying existing master\r\ndebug1:
 Control socket \"/home/student/.ansible/cp/d4775f48c9\" does not exist\r\nnssh:
```

```
Could not resolve hostname servera.lab.exammple.com: Name or service not known\r
\n", "unreachable": true }
... Output omitted ...
PLAY RECAP ****
servera.lab.exammple.com : ok=0 changed=0 unreachable=1 failed=0
```

15. When using the highest level of verbosity with Ansible, the Ansible log file is a better option to check output than the console. Check the output from the previous command in the `/home/student/troubleshooting/ansible.log` file.

```
[student@workstation troubleshooting]$ tail ansible.log
... Output omitted ...
2017-07-11 03:51:56,460 p=3333 u=student | Using module file /usr/lib/python2.7/
site-packages/ansible/modules/system/setup.py
2017-07-11 03:51:56,531 p=3333 u=student | fatal: [servera.lab.exammple.com]:
UNREACHABLE! => {
 "changed": false,
 "msg": "Failed to connect to the host via ssh: OpenSSH_6.6.1, OpenSSL 1.0.1e-
fips 11 Feb 2013\r\ndebug1: Reading configuration data /home/student/.ssh/config
\r\ndebug1: /home/student/.ssh/config line 1: Applying options for *\r\ndebug1:
Reading configuration data /etc/ssh/ssh_config\r\ndebug1: /etc/ssh/ssh_config line
56: Applying options for *\r\ndebug1: auto-mux: Trying existing master\r\ndebug1:
Control socket \"/home/student/.ansible/cp/d4775f48c9\" does not exist\r\nssh:
Could not resolve hostname servera.lab.exammple.com: Name or service not known\r
\n",
 "unreachable": true
}
2017-07-11 03:51:56,537 p=3333 u=student | to retry, use: --limit @/home/
student/troubleshooting/samba.retry

2017-07-11 03:51:56,538 p=3333 u=student | PLAY RECAP ****
2017-07-11 03:51:56,538 p=3333 u=student | servera.lab.exammple.com : ok=0
changed=0 unreachable=1 failed=0
```

16. Investigate the `inventory` file for errors. Notice the `[samba_servers]` group has misspelled `servera.lab.example.com`. Correct it and make the file look like the following:

```
... Output omitted ...
[samba_servers]
servera.lab.example.com
... Output omitted ...
```

17. Run the playbook again. The `debug install_state variable` task returns the message *The state for the samba service is installed*. This task makes use of the `debug` module, and displays the value of the `install_state` variable. An error is also shown in the `deliver samba config` task, since no `samba.j2` file is available in the working directory, `/home/student/troubleshooting/`.

```
[student@workstation troubleshooting]$ ansible-playbook samba.yml

PLAY [Install a samba server] ****
... Output omitted ...
TASK [debug install_state variable] ****
ok: [servera.lab.example.com] => {
 "msg": "The state for the samba service is installed"
```

```
}

... Output omitted ...
TASK [deliver samba config] ****
fatal: [servera.lab.example.com]: FAILED! => {"changed": false, "failed": true,
 "msg": "Unable to find 'samba.j2' in expected paths."}
... Output omitted ...
PLAY RECAP ****
servera.lab.example.com : ok=7 changed=3 unreachable=0 failed=1
```

18. Edit the playbook, and correct the **src** parameter in the *deliver samba config* task to be **samba.conf.j2**. When you are finished it should look like the following:

```
... Output omitted ...
- name: deliver samba config
 template:
 src: samba.conf.j2
 dest: /etc/samba/smb.conf
 owner: root
... Output omitted ...
```

19. Run the playbook again. Execute the playbook using the **--step** option. It should run without errors.

```
[student@workstation troubleshooting]$ ansible-playbook samba.yml --step

PLAY [Install a samba server] ****
Perform task: TASK: setup (y/n/c): y
... Output omitted ...
Perform task: TASK: install samba (y/n/c): y
... Output omitted ...
Perform task: TASK: install firewalld (y/n/c): y
... Output omitted ...
Perform task: TASK: debug install_state variable (y/n/c): y
... Output omitted ...
Perform task: TASK: start samba (y/n/c): y
... Output omitted ...
Perform task: TASK: start firewalld (y/n/c): y
... Output omitted ...
Perform task: TASK: configure firewall for samba (y/n/c): y
... Output omitted ...
Perform task: TASK: deliver samba config (y/n/c): y
... Output omitted ...
PLAY RECAP ****
servera.lab.example.com : ok=8 changed=1 unreachable=0 failed=0
```

# Troubleshooting Ansible Managed Hosts

## Objectives

After completing this section, students should be able to:

- Use the **ansible-playbook --check** command
- Use modules to probe service status on the managed hosts
- Use ad hoc commands to check issues on the managed hosts

## Check Mode as a Testing Tool

You can use the **ansible-playbook --check** command to run smoke tests on a playbook. This option executes the playbook without making any change to the managed hosts' configuration. If a module used within the playbook supports *check mode* the changes that would be made in the managed hosts are displayed. If that mode is not supported by a module those changes will not be displayed.

```
[student@demo ~]# ansible-playbook --check playbook.yml
```

The **ansible-playbook --check** command is used when all the tasks included in a playbook have to be executed in *check mode*, but if just some of those tasks need to be executed in *check mode*, the **always\_run** option is a better solution. Each task can have a **always\_run** clause associated. The value for this clause is *true* if the task has to be executed in check mode or *false* if it is not. The following task is executed in check mode.

```
tasks:
 - name: task in check mode
 shell: uname -a
 always_run: yes
```

The following task is not executed in check mode.

```
tasks:
 - name: task in check mode
 shell: uname -a
 always_run: false
```



### Note

The **ansible-playbook --check** command may not work properly if tasks make use of conditionals.

Ansible also provides the **--diff** option. This option reports the changes done to the template files on managed hosts. If used with the **--check** option, those changes are displayed and not actually made.

```
[student@demo ~]# ansible-playbook --check --diff playbook.yml
```

## Modules for testing

Some modules can provide additional information about what the status of a managed host is. The following list includes some of the Ansible modules that can be used to test and debug issues on managed hosts.

- The **uri** module provides a way to check that a RESTful API is returning the required content.

```
tasks:
 - uri:
 url: http://api.myapp.com
 return_content: yes
 register: apiresponse

 - fail:
 msg: 'version was not provided'
 when: "'version' not in apiresponse.content"
```

- The **script** module supports the execution of a script on a managed host, failing if the return code for that script is non-zero. The script must be on the control node and will be transferred (and executed) on the managed host.

```
tasks:
 - script: check_free_memory
```

- The **stat** module can check that files and directories not managed directly by Ansible are present. Check the usage of the **assert** module in the following example checking that a file exists in the managed host.

```
tasks:
 - stat:
 path: /var/run/app.lock
 register: lock

 - assert:
 that:
 - lock.stat.exists
```

## Using ad hoc commands for testing

The following examples illustrate some of the checks that can be done on a managed host through the use of ad hoc commands. Those examples use modules such as *yum* in order to perform checks on the managed nodes. This example checks that the *httpd* package is currently installed in the **demohost** managed host.

```
[student@demo ~]# ansible demohost -u devops -b -m yum -a 'name=httpd state=present'
```

This example returns the currently available space on the disks configured in the **demohost** managed host.

```
[student@demo ~]# ansible demohost -a 'lsblk'
```

This example returns the currently available free memory on the **demohost** managed host.

```
[student@demo ~]# ansible demohost -a 'free -m'
```

## The correct level of testing

Ansible ensures that the configuration included in playbooks and performed by its modules is correctly done. It monitors all modules for reported failures, and stops the playbook immediately any failure is encountered. This ensures that any task performed before the failure has no errors. Because of this, there is no need to check if the result of a task managed by Ansible has been correctly applied on the managed hosts. It makes sense to add some health checks either to playbooks, or run those directly as ad hoc commands, when more direct troubleshooting is required.



### References

- Check Mode ("Dry Run") -- Ansible Documentation  
[http://docs.ansible.com/ansible/playbooks\\_checkmode.html](http://docs.ansible.com/ansible/playbooks_checkmode.html)
- Testing Strategies -- Ansible Documentation  
[http://docs.ansible.com/ansible/test\\_strategies.html](http://docs.ansible.com/ansible/test_strategies.html)

# Guided Exercise: Troubleshooting Ansible Managed Hosts

In this exercise, the SMTP service is deployed in a managed host. The playbook for this exercise is the `mailrelay.yml` playbook that configures an SMTP service on `servera.lab.example.com`.

## Outcomes

You should be able to:

- Troubleshoot managed hosts.

## Before you begin

Log in to `workstation` as `student` using `student` as the password.

On `workstation`, run the `lab troubleshoot-managedhosts setup` script. It checks if Ansible is installed on `workstation`. It also downloads the `inventory`, `mailrelay.yml`, and `postfix-relay-main.conf.j2` files from <http://materials.example.com/troubleshooting/> to the `/home/student/troubleshooting/` directory.

```
[student@workstation ~]$ lab troubleshoot-managedhosts setup
```

## Steps

- On workstation, change to the `/home/student/troubleshooting/` directory.

```
[student@workstation ~]$ cd ~/troubleshooting/
```

- Run the `mailrelay.yml` playbook using check mode.

```
[student@workstation troubleshooting]$ ansible-playbook mailrelay.yml --check
PLAY [create mail relay servers] ****
...
TASK [check main.cf file] ****
ok: [servera.lab.example.com]

TASK [verify main.cf file exists] ****
ok: [servera.lab.example.com] => {
 "msg": "The main.cf file exists"
}
...
TASK [email notification of always_bcc config] ****
fatal: [servera.lab.example.com]: FAILED! => {"failed": true, "msg": "The conditional check 'bcc_state.stdout != 'always_bcc =' failed. The error was: error while evaluating conditional (bcc_state.stdout != 'always_bcc ='): 'dict object' has no attribute 'stdout'\n\nThe error appears to have been in '/home/student/troubleshooting/mailrelay.yml': line 42, column 7, but may\nbe elsewhere in the file depending on the exact syntax problem.\n\nThe offending line appears to be:\n- name: email notification of always_bcc config\n ^ here\n"}
...
PLAY RECAP ****
servera.lab.example.com : ok=6 changed=1 unreachable=0 failed=1
```

The *verify main.cf file exists* task uses the *stat* module. It confirmed that **main.cf** exists on **servera.lab.example.com**.

The *email notification of always\_bcc config* task failed. It did not receive output from the *check for always\_bcc* task, because the playbook was executed using check mode.

3. Using an ad hoc command, check the header for the **/etc/postfix/main.cf** file.

```
[student@workstation troubleshooting]$ ansible servera.lab.example.com -u devops -b
-a "head /etc/postfix/main.cf"
servera.lab.example.com | FAILED | rc=1 >>
head: cannot open '/etc/postfix/main.cf' for reading: No such file or directory
```

The command failed because the playbook was executed using check mode. Postfix is not installed on **servera.lab.example.com**

4. Run the playbook again, but without specifying check mode. The error in the *email notification of always\_bcc config* task should disappear.

```
[student@workstation troubleshooting]$ ansible-playbook mailrelay.yml
PLAY [create mail relay servers] *****
...
TASK [check for always_bcc] *****
changed: [servera.lab.example.com]

TASK [email notification of always_bcc config] *****
skipping: [servera.lab.example.com]

RUNNING HANDLER [restart postfix] *****
changed: [servera.lab.example.com]

PLAY RECAP *****
servera.lab.example.com : ok=8 changed=3 unreachable=0 failed=0
```

5. Using an ad hoc command, display the top of the **/etc/postfix/main.cf** file.

```
[student@workstation troubleshooting]$ ansible servera.lab.example.com -u devops -b
-a "head /etc/postfix/main.cf"
servera.lab.example.com | SUCCESS | rc=0 >>
Ansible managed

Global Postfix configuration file. This file lists only a subset
of all parameters. For the syntax, and for a complete parameter
list, see the postconf(5) manual page (command: "man 5 postconf").

For common configuration examples, see BASIC_CONFIGURATION_README
and STANDARD_CONFIGURATION_README. To find these documents, use
the command "postconf html_directory readme_directory", or go to
http://www.postfix.org/.
```

Now it starts with a line that contains the string, "Ansible managed". This file was updated and is now managed by Ansible.

6. Add a task to enable the *smtp* service through the firewall.

```
[student@workstation troubleshooting]$ vim mailrelay.yml
```

```

...
 - name: postfix firewalld config
 firewalld:
 state: enabled
 permanent: true
 immediate: true
 service: smtp
...

```

7. Run the playbook. The *postfix firewalld config* should have been executed with no errors.

```

[student@workstation troubleshooting]$ ansible-playbook mailrelay.yml
PLAY [create mail relay servers] ****
...
TASK [postfix firewalld config] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com : ok=8 changed=2 unreachable=0 failed=0

```

8. Using an ad hoc command, check that the *smtp* service is now configured in the firewall at **servera.lab.example.com**.

```

[student@workstation troubleshooting]$ ansible servera.lab.example.com -u devops -b
-a "firewall-cmd --list-services"
servera.lab.example.com | SUCCESS | rc=0 >>
dhcpv6-client samba smtp ssh

```

9. Test the SMTP service, listening on port *TCP/25*, on **servera.lab.example.com** with **telnet**. Disconnect when you are finished.

```

[student@workstation troubleshooting]$ telnet servera.lab.example.com 25
Trying 172.25.250.10...
Connected to servera.lab.example.com.
Escape character is '^].
220 servera.lab.example.com ESMTP Postfix
quit
Connection closed by foreign host.

```

# Lab: Troubleshooting Ansible

In this lab, the control node and the managed hosts have errors that need to be corrected. The **secure-web.yml** playbook configures Apache with SSL on **serverb.lab.example.com**.

## Outcomes

You should be able to:

- Troubleshoot playbooks.
- Troubleshoot managed hosts.

## Before you begin

Log in to **workstation** as **student** using **student** as the password. Run the **lab troubleshoot-lab setup** command.

```
[student@workstation ~]$ lab troubleshoot-lab setup
```

This script checks if Ansible is installed on **workstation**, creates the **~student/troubleshooting-lab/** directory, and the **html** subdirectory in it. It also downloads from **http://materials.example.com/troubleshooting/** the **ansible.cfg**, **inventory-lab**, **secure-web.yml**, and **vhosts.conf** files to the **/home/student/troubleshooting-lab/** directory, and the **index.html** file to the **/home/student/troubleshooting-lab/html/** directory.

## Steps

1. From the **~/troubleshooting-lab** directory, run the **secure-web.yml** playbook. It uses by default the **inventory-lab** file configured in the **inventory** parameter of the **ansible.cfg** file. This playbook sets up Apache with SSL. Solve any syntax issues in the variables definition.
2. Rerun the playbook and solve any issues related to the indentation in the **secure-web** playbook.
3. Rerun the playbook and solve any issues related to variable quotation in the **secure-web** playbook.
4. Rerun the playbook and solve any issues related to the inventory.
5. Rerun the playbook and solve any issues related to the user used to connect to the managed hosts.
6. Rerun the playbook and solve any issues related to the ability of the **devops** user to escalate the privilege to root.
7. Ensure that the Apache service playbook has been executed successfully on **serverb.lab.example.com**. Try to run the playbook first using check mode and check the state of the **httpd** service on **serverb.lab.example.com** using an ad hoc command. Run the playbook again and recheck the service.

## Evaluation

From **workstation**, run the **lab troubleshoot-lab grade** script to confirm success on this exercise.

```
[student@workstation troubleshooting-lab]$ lab troubleshoot-lab grade
```

#### Cleanup

From **workstation**, run the **lab troubleshoot-lab cleanup** script to clean up this lab.

```
[student@workstation troubleshooting-lab]$ lab troubleshoot-lab cleanup
```

# Solution

In this lab, the control node and the managed hosts have errors that need to be corrected. The **secure-web.yml** playbook configures Apache with SSL on **serverb.lab.example.com**.

## Outcomes

You should be able to:

- Troubleshoot playbooks.
- Troubleshoot managed hosts.

## Before you begin

Log in to **workstation** as **student** using **student** as the password. Run the **lab troubleshoot-lab setup** command.

```
[student@workstation ~]$ lab troubleshoot-lab setup
```

This script checks if Ansible is installed on **workstation**, creates the **~student/troubleshooting-lab/** directory, and the **html** subdirectory in it. It also downloads from **http://materials.example.com/troubleshooting/** the **ansible.cfg**, **inventory-lab**, **secure-web.yml**, and **vhosts.conf** files to the **/home/student/troubleshooting-lab/** directory, and the **index.html** file to the **/home/student/troubleshooting-lab/html/** directory.

## Steps

1. From the **~/troubleshooting-lab** directory, run the **secure-web.yml** playbook. It uses by default the **inventory-lab** file configured in the **inventory** parameter of the **ansible.cfg** file. This playbook sets up Apache with SSL. Solve any syntax issues in the variables definition.
  - 1.1. On workstation, change to the **/home/student/troubleshooting-lab** project directory.

```
[student@workstation ~]$ cd ~/troubleshooting-lab/
```

- 1.2. Run the **secure-web** playbook, included in the **secure-web.yml** file. This playbook sets up Apache with SSL if everything is correct.

```
[student@workstation troubleshooting-lab]$ ansible-playbook secure-web.yml
ERROR! Syntax Error while loading YAML.
...
The error appears to have been in '/home/student/Ansible-course/troubleshooting-lab/secure-web.yml': line 7, column 30, but may
be elsewhere in the file depending on the exact syntax problem.

The offending line appears to be:

vars:
 random_var: This is colon: test
 ^ here
```

- 1.3. Correct the syntax issue in the variables definition by adding double quotes to the **This is colon: test** string. The resulting change should look like the following:

```
... Output omitted ...
vars:
 random_var: "This is colon: test"
... Output omitted ...
```

2. Rerun the playbook and solve any issues related to the indentation in the *secure-web* playbook.

- 2.1. Rerun the *secure-web* playbook, included in the **secure-web.yml** file, using the **--syntax-check** option.

```
[student@workstation troubleshooting-lab]$ ansible-playbook secure-web.yml --syntax-check
```

```
ERROR! Syntax Error while loading YAML.
```

The error appears to have been in '/home/student/Ansible-course/troubleshooting-lab/secure-web.yml': line 43, column 10, but may be elsewhere in the file depending on the exact syntax problem.

The offending line appears to be:

```
- name: start and enable web services
^ here
```

- 2.2. Correct any syntax issues in the indentation. Remove the extra space at the beginning of the *start and enable web services* task elements. The resulting change should look like the following:

```
... Output omitted ...
- name: start and enable web services
 service:
 name: httpd
 state: started
 enabled: yes
 tags:
 - services
... Output omitted ...
```

3. Rerun the playbook and solve any issues related to variable quotation in the *secure-web* playbook.

- 3.1. Rerun the *secure-web* playbook, included in the **secure-web.yml** file, using the **--syntax-check** option.

```
[student@workstation troubleshooting-lab]$ ansible-playbook secure-web.yml --syntax-check
```

```
ERROR! Syntax Error while loading YAML.
```

The error appears to have been in '/home/student/Ansible-course/troubleshooting-lab/secure-web.yml': line 13, column 20, but may be elsewhere in the file depending on the exact syntax problem.

The offending line appears to be:

```
 yum:
 name: {{ item }}
 ^ here
```

We could be wrong, but this one looks like it might be an issue with missing quotes. Always quote template expression brackets when they start a value. For instance:

```
 with_items:
 - {{ foo }}
```

Should be written as:

```
 with_items:
 - "{{ foo }}"
```

- 3.2. Correct the *item* variable in the **install web server packages** task. Add double quotes to **{{ item }}**. The resulting change should look like the following:

```
... Output omitted ...
 - name: install web server packages
 yum:
 name: "{{ item }}"
 state: latest
 notify:
 - restart services
 tags:
 - packages
 with_items:
 - httpd
 - mod_ssl
 - crypto-utils
... Output omitted ...
```

- 3.3. Rerun the playbook using the **--syntax-check** option. It should not show any additional syntax errors.

```
[student@workstation troubleshooting-lab]$ ansible-playbook secure-web.yml --syntax-check
playbook: secure-web.yml
```

4. Rerun the playbook and solve any issues related to the inventory.

- 4.1. Rerun the *secure-web* playbook, included in the **secure-web.yml** file.

```
[student@workstation troubleshooting-lab]$ ansible-playbook secure-web.yml
PLAY [create secure web service] ****
TASK [Gathering Facts] ****
fatal: [serverb.lab.example.com]: UNREACHABLE! => {"changed": false,
 "msg": "Failed to connect to the host via ssh: Warning: Permanently added
 'tower.lab.example.com,172.25.250.9' (ECDSA) to the list of known hosts.\r
 \nPermission denied (publickey,gssapi-keyex,gssapi-with-mic,password).\r\n",
 "unreachable": true}
 to retry, use: --limit @/home/student/troubleshooting-lab/secure-
web.retry
```

```
PLAY RECAP ****
serverb.lab.example.com : ok=0 changed=0 unreachable=1 failed=0
```

- 4.2. Rerun the *secure-web* playbook again with *-vvvv* parameter to add verbosity. Notice that the connection appears to be to **tower.lab.example.com**, instead of **serverb.lab.example.com**.

```
[student@workstation troubleshooting-lab]$ ansible-playbook secure-web.yml -vvvv
TASK [Gathering Facts] ****
tower.lab.example.com ESTABLISH SSH CONNECTION FOR USER: students
tower.lab.example.com SSH: EXEC ssh -C -vvv -o ControlMaster=auto
-o ControlPersist=60s -o Port=22 -o KbdInteractiveAuthentication=no
-o PreferredAuthentications=gssapi-with-mic,gssapi-keyex,hostbased,publickey
-o PasswordAuthentication=no -o User=students -o ConnectTimeout=10
-o ControlPath=/home/student/.ansible/cp/ansible-ssh-%C -tt
tower.lab.example.com
'/bin/sh -c """(umask 22 && mkdir -p "` echo $HOME/.ansible/tmp/
ansible- tmp-1460241127.16-3182613343880 `" && echo "` echo $HOME/.ansible/
tmp/ansible-tmp-1460241127.16-3182613343880 `")""""'
```

- 4.3. Correct the inventory. Delete the **ansible\_host** host variable so the file looks like the following:

```
[webservers]
serverb.lab.example.com
```

5. Rerun the playbook and solve any issues related to the user used to connect to the managed hosts.

- 5.1. Rerun the *secure-web* playbook, included in the **secure-web.yml** file.

```
[student@workstation troubleshooting-lab]$ ansible-playbook secure-web.yml -vvvv
...
TASK [Gathering Facts]

serverb.lab.example.com ESTABLISH SSH CONNECTION FOR USER: students
serverb.lab.example.com EXEC ssh -C -vvv -o ControlMaster=auto
-o ControlPersist=60s -o Port=22 -o KbdInteractiveAuthentication=no
-o PreferredAuthentications=gssapi-with-mic,gssapi-keyex,hostbased,publickey
-o PasswordAuthentication=no -o User=students -o ConnectTimeout=10
-o ControlPath=/home/student/.ansible/cp/ansible-ssh-%C -tt
serverb.lab.example.com '/bin/sh -c """(umask 22 && mkdir -p "` echo $HOME/.ansible/tmp/ansible-tmp-1460241127.16-3182613343880 `" &&
echo "` echo $HOME/.ansible/tmp/ansible-tmp-1460241127.16-3182613343880 `")""""'
... Output omitted ...
```

- 5.2. Edit the *secure-web* playbook to specify the **devops** user. The first lines of the playbook should look like the following:

```

start of secure web server playbook
- name: create secure web service
 hosts: webservers
 user: devops
```

... Output omitted ...

6. Rerun the playbook and solve any issues related to the ability of the *devops* user to escalate the privilege to root.

- 6.1. Rerun the *secure-web* playbook, included in the **secure-web.yml** file.

```
[student@workstation troubleshooting-lab]$ ansible-playbook secure-web.yml -vvvv
...
failed: [serverb.lab.example.com] => (item=[u'httpd', u'mod_ssl',
u'crypto-utils']) => {"changed": true, "failed": true, "invocation":
{"module_args": {"conf_file": null, "disable_gpg_check": false, "disablerepo": null,
"enablerepo": null, "exclude": null, "install_repoquery": true,
"list": null, "name": ["httpd", "mod_ssl", "crypto-utils"], "state": "latest",
"update_cache": false}, "module_name": "yum"}, "item": ["httpd", "mod_ssl",
"crypto-utils"], "msg": "You need to be root to perform this command.\n",
"rc": 1, "results": ["Loaded plugins: langpacks, search-disabled-repos\n"]}
...
Output omitted ...
```

- 6.2. Edit the playbook to include the **become** parameter. The resulting change should look like the following:

```

start of secure web server playbook
- name: create secure web service
 hosts: webservers
 user: devops
 become: true
...
Output omitted ...
```

7. Ensure that the Apache service playbook has been executed successfully on **serverb.lab.example.com**. Try to run the playbook first using check mode and check the state of the **httpd** service on **serverb.lab.example.com** using an ad hoc command. Run the playbook again and recheck the service.
- 7.1. Rerun the **secure-web.yml** playbook in check mode. The **install web server packages** task reflects the changes that would be done. The **httpd\_conf\_syntax variable** task shows the current value for the **httpd\_conf\_syntax** variable.

```
[student@workstation troubleshooting-lab]$ ansible-playbook secure-web.yml --check
PLAY [create secure web service] ****
...
TASK [install web server packages] ****
changed: [serverb.lab.example.com] => (item=[u'httpd', u'mod_ssl', u'crypto-
utils'])
...
TASK [httpd_conf_syntax variable] ****
ok: [serverb.lab.example.com] => {
 "msg": "The httpd_conf_syntax variable value is {u'msg': u'remote module
does not support check mode', u'skipped': True, u'changed': False}"
}
...
RUNNING HANDLER [restart services] ****
changed: [serverb.lab.example.com]
...
PLAY RECAP ****
```

```
serverb.lab.example.com : ok=6 changed=5 unreachable=0 failed=0
```

- 7.2. Using an ad hoc command, check the state of the *httpd* service in **serverb.lab.example.com**. The *httpd* service is not installed in **serverb.lab.example.com**.

```
[student@workstation troubleshooting-lab]$ ansible all -u devops -b -a
'systemctl status httpd'
serverb.lab.example.com | FAILED | rc=4 >>
Unit httpd.service could not be found.
```

- 7.3. Rerun the *secure-web* playbook, included in the **secure-web.yml** file.

```
[student@workstation troubleshooting-lab]$ ansible-playbook secure-web.yml
PLAY [create secure web service] ****
...
TASK [install web server packages] ****
changed: [serverb.lab.example.com] => (item=[u'httpd', u'mod_ssl', u'crypto-
utils'])
...
TASK [httpd_conf_syntax variable] ****
ok: [serverb.lab.example.com] => {
 "msg": "The httpd_conf_syntax variable value is {u'changed': True, u'end':
u'2016-05-03 19:43:37.612170', u'stdout': u'', u'cmd': [u'/sbin/httpd', u'-t'],
'failed': False, u'delta': u'0:00:00.033463', u'stderr': u'Syntax OK', u'rc':
0, 'stdout_lines': [], 'failed_when_result': False, u'start': u'2016-05-03
19:43:37.578707', u'warnings': []}"
}
...
RUNNING HANDLER [restart services] ****
changed: [serverb.lab.example.com]

PLAY RECAP ****
serverb.lab.example.com : ok=10 changed=7 unreachable=0 failed=0
```

- 7.4. Using an ad hoc command, check the state of the *httpd* service in **serverb.lab.example.com**. The *httpd* service should be now running in **serverb.lab.example.com**.

```
[student@workstation troubleshooting-lab]$ ansible all -u devops -b -a
'systemctl status httpd'
serverb.lab.example.com | SUCCESS | rc=0 >>
• httpd.service - The Apache HTTP Server
 Loaded: loaded (/usr/lib/systemd/system/httpd.service; enabled; vendor
 preset: disabled)
 Active: active (running) since Tue 2016-05-03 19:43:39 CEST; 12s ago
 ... Output omitted ...
```

### Evaluation

From **workstation**, run the **lab\_troubleshoot-lab grade** script to confirm success on this exercise.

```
[student@workstation troubleshooting-lab]$ lab_troubleshoot-lab grade
```

**Cleanup**

From **workstation**, run the **lab troubleshoot-lab cleanup** script to clean up this lab.

```
[student@workstation troubleshooting-lab]$ lab troubleshoot-lab cleanup
```

## Summary

In this chapter, you learned:

- Ansible provides built-in logging. This feature is not enabled by default.
- The **log\_path** parameter in the **default** section of the **ansible.cfg** configuration file specifies the location of the log file to which all Ansible output is redirected.
- The **debug** module provides additional debugging information while running a playbook (for example, current value for a variable).
- The **-v** option of the **ansible-playbook** command provides several levels of output verbosity. This is useful for debugging Ansible tasks when running a playbook.
- The **--check** option enables Ansible modules with check mode support to display changes to be performed, instead of applying those changes to the managed hosts.
- Additional checks can be executed on the managed hosts using ad hoc commands.
- There is no need to double-check the configuration performed by Ansible as long as the playbook completes successfully.



## CHAPTER 11

# IMPLEMENTING ANSIBLE TOWER

Overview	
<b>Goal</b>	Explain what Ansible Tower is and demonstrate a basic ability to navigate and use its web user interface.
<b>Objectives</b>	<ul style="list-style-type: none"><li>Describe the architecture, use cases, and installation requirements of Ansible Tower.</li><li>Install a new Ansible Tower on a single node using the setup.sh script.</li><li>Navigate and describe the Ansible Tower web user interface, and successfully launch a job using the demo job template, project, credential, and inventory.</li></ul>
<b>Sections</b>	<ul style="list-style-type: none"><li>Introduction to Ansible Tower (and Quiz)</li><li>Installing Ansible Tower (and Guided Exercise)</li><li>Navigating the Ansible Tower Web Interface (and Guided Exercise)</li></ul>
<b>Quiz</b>	<ul style="list-style-type: none"><li>Architecture and Installation of Ansible Tower</li></ul>

# Introduction to Ansible Tower

## Objectives

After completing this section, students should be able to describe the architecture, use cases, and installation requirements of Ansible Tower.

## Why Ansible Tower?

As an enterprise's experience with Ansible matures, it often finds additional opportunities for leveraging Ansible to simplify and improve IT operations. The same Ansible playbooks utilized by operations teams to deploy production systems can also be used to deploy identical systems in earlier stages of the software development lifecycle. When automated with Ansible, complex production support tasks typically handled by skilled engineers can easily be delegated to and resolved by entry-level technicians.

However, sharing an existing Ansible infrastructure to scale IT automation across an enterprise can present some challenges. While properly written Ansible playbooks can be leveraged across teams, Ansible does not provide any facilities for managing their shared access. Additionally, though playbooks may allow for the delegation of complex tasks, their execution may require highly privileged and guarded administrator credentials.

IT teams often vary in their preferred tool sets. While some may prefer the direct execution of playbooks, other teams may wish to trigger playbook execution from existing continuous integration and delivery tool suites. In addition, those that traditionally work with GUI-based tools may find Ansible's pure command-line interface intimidating and awkward.

Ansible Tower overcomes many of these problems by providing a framework for running and managing Ansible efficiently on an enterprise scale. Tower eases the administration involved with sharing an Ansible infrastructure while maintaining organization security by introducing features such as a centralized web interface for playbook management, *role-based access control (RBAC)*, and centralized logging and auditing. Its REST API ensures that Tower integrates easily with an enterprise's existing workflows and tool sets. Tower's API and notification features make it particularly easy to associate Ansible playbooks with other tools such as Jenkins, CloudForms, or Red Hat Satellite, to enable continuous integration and deployment. It provides mechanisms to enable centralized use and control of machine credentials and other secrets without exposing them to end users of Ansible Tower.

## Ansible Tower Architecture

Ansible Tower is a Django web application designed to run on a Linux server as an on-premise, self-hosted solution which layers on top of an enterprise's existing Ansible infrastructure.

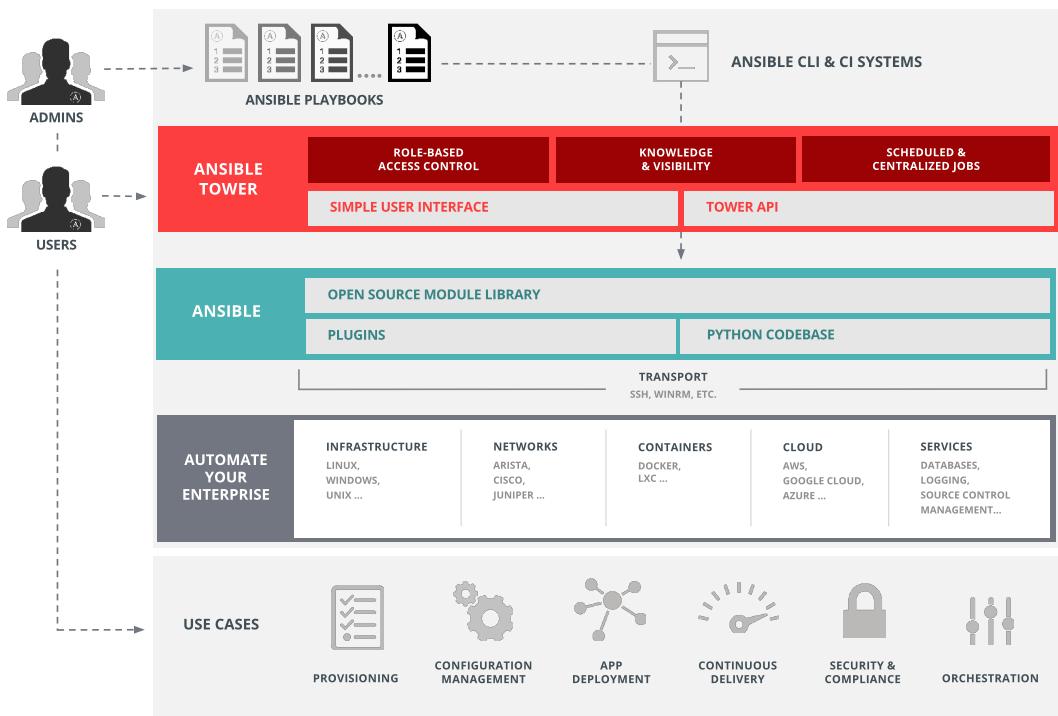


Figure 11.1: Ansible Tower architecture

Users interface with their enterprise's underlying Ansible infrastructure through either Tower's web interface or its RESTful API. Tower's web interface is a graphical interface wrapper which performs its actions by executing calls against the Tower API. Any action available through the Tower web interface can therefore also be performed through Tower's RESTful API. The RESTful API is essential for those users looking to integrate Ansible with existing software tools and processes.

Tower stores its data in a PostgreSQL back-end database and makes use of the RabbitMQ messaging system. Versions of Ansible Tower prior to version 3.0 also relied on a MongoDB database. This dependency has since been removed and data is now stored solely in a PostgreSQL database.

Depending on an enterprise's needs, Ansible Tower can be implemented using one of the following architectures.

#### Single Machine with Integrated Database

All Tower components, the web front end, RESTful API back end, and PostgreSQL database, reside on a single machine. This is the standard architecture.

#### Single Machine with Remote Database

Tower web front end and RESTful API back end are installed on a single machine while the PostgreSQL database is installed remotely on another server on the same network. The remote database can be hosted on a server with an existing PostgreSQL instance outside the management of Tower. Another option is to have the Tower installer create a Tower-managed PostgreSQL instance on the remote server and populate it with the Tower database.

### High Availability Multi-Machine Cluster

Older Tower versions offered a redundant, active-passive Tower architecture consisting of a single active node and one or more inactive nodes. Starting with Tower 3.1, this architecture is now replaced by an active-active, high-availability cluster consisting of multiple active tower nodes.

Each node in the cluster hosts the Tower web front end and RESTful API back end, and can receive and process requests. In this cluster architecture, the PostgreSQL database is hosted on a remote server. The remote database can reside either on a server with an existing PostgreSQL instance outside the management of Tower or on a server with a Tower-managed PostgreSQL instance created by the Tower installer.



#### Note

This course focuses on the most straightforward architecture to deploy, a single Ansible Tower server with an integrated database.

## Ansible Tower features

Two types of license are available for Ansible Tower: basic and enterprise. Enterprise license offers access to all Tower features. Basic license offers access to only a subset of Tower's features and does not include many enterprise-level options, such as system tracking, logging aggregation, and clustering.

The following are just some of the many features offered by Ansible Tower for controlling, securing, and managing Ansible in an enterprise environment.

### Visual Dashboard

The Tower web interface opens into a dashboard screen which provides a summary view of an enterprise's entire Ansible environment. The Tower Dashboard allows administrators to easily see the current status of hosts and inventories, as well as the results of recent job executions.

### Role-based Access Control (RBAC)

Tower utilizes a Role-Based Access Control (RBAC) system which maintains security while streamlining user access management. It simplifies the delegation of user access to Tower objects such as Organizations, Projects, and Inventories.

### Graphical Inventory Management

Tower offers users the ability to create inventory groups and add inventory hosts through its web interface. Inventories can also be updated from an external inventory source such as from public cloud providers, local virtualization environments, and an organization's custom *configuration management database (CMDB)*.

### Job Scheduling

Tower offers users the ability to schedule playbook execution and updates from external data sources either on a one-time basis or to be repeated at regular intervals. This allows routine tasks to be performed unattended and is especially useful for tasks such as backup routines which should ideally be executed during operational off-hours.

### Real-Time and Historical Job Status Reporting

When playbook executions are initiated in Tower, the web interface displays the playbook's output and execution results in real time. Results of previously executed jobs and scheduled job runs are also made available by Tower.

### Push-Button Automation

Ansible simplifies IT automation while Tower takes it a step further by enabling user self-service. Tower's streamlined web interface, coupled with the flexibility of its RBAC system, allows administrators to safely delegate complex tasks as single click-of-a-button routines.

### Remote Command Execution

Tower makes the on-demand flexibility of Ansible's ad-hoc commands available through its remote command execution feature. User permissions for remote command execution is enforced using Tower's RBAC system.

### Credential Management

Tower centrally manages the credentials which are used for authentication purposes: to do things like running Ansible plays on managed hosts, synchronizing information from dynamic inventory sources, and importing Ansible project content from version control systems. It encrypts the passwords or keys provided so that they can not be retrievable by Tower users. Users can be granted the ability to use or replace these credentials without actually exposing them to the user.

### Centralized Logging and Auditing

All playbook and remote command executions initiated on Tower are logged. This provides the ability to audit when each job was executed and by whom. In addition, Tower offers the ability to integrate its log data into third party logging aggregation solutions, such as Splunk and Sumologic.

### Integrated Notifications

Tower Notifications can be used to signal when Tower job executions succeed or fail. Notifications can be delivered using many different protocols, including email, Slack, and HipChat.

### Multi-Playbook Workflows

Complex operations often involve the serial execution of multiple playbooks. Tower's multi-playbook workflows allow users to chain together multiple playbooks to facilitate the execution of complex routines involving provisioning, configuration, deployment, and orchestration. An intuitive workflow editor also helps to simplify the modeling of multi-playbook workflows.

### System Tracking

Tower can be configured to routinely scan managed hosts and record their states. The collected data can be used to audit system changes over time. Additionally, this feature can be used to compare and detect differences between systems.

### RESTful API

Tower's RESTful API exposes every Tower feature available through Tower's web interface. The API's browsable format makes it self-documenting and simplifies lookup of API usage information.



## References

*Ansible Tower Administration Guide* for Ansible Tower 3.1.1  
| <http://docs.ansible.com/ansible-tower/3.1.1/html/administration>

## Quiz: Introduction to Ansible Tower

Choose the correct answer(s) to the following questions:

1. Which of the following three features are provided by Ansible Tower? (Choose three.)
  - a. Role-based access control
  - b. Playbook creator wizard
  - c. Centralized logging
  - d. RESTful API
  
2. Which of the following two enhancements are **additions** to Ansible provided by Ansible Tower? (Choose two.)
  - a. Playbook development
  - b. Remote execution
  - c. Multi-play workflows
  - d. Monitoring
  - e. Version Control
  - f. Graphical inventory management
  
3. Which of the following three architectures are supported by the Tower installer? (Choose three.)
  - a. Single machine with integrated database
  - b. Single machine with an external database hosted on a separate server on the network.
  - c. High-availability, multi-machine cluster with an external database
  - d. Active/passive redundancy multi-machine with an external database
  
4. Which of the following two features are not provided by a basic Ansible Tower license? (Choose two.)
  - a. Tower dashboard
  - b. Job scheduling
  - c. System tracking
  - d. Role-based access control
  - e. Logging aggregation

## Solution

Choose the correct answer(s) to the following questions:

1. Which of the following three features are provided by Ansible Tower? (Choose three.)
  - a. **Role-based access control**
  - b. *Playbook creator wizard*
  - c. **Centralized logging**
  - d. *RESTful API*
2. Which of the following two enhancements are ***additions*** to Ansible provided by Ansible Tower? (Choose two.)
  - a. *Playbook development*
  - b. *Remote execution*
  - c. **Multi-play workflows**
  - d. *Monitoring*
  - e. *Version Control*
  - f. **Graphical inventory management**
3. Which of the following three architectures are supported by the Tower installer? (Choose three.)
  - a. **Single machine with integrated database**
  - b. **Single machine with an external database hosted on a separate server on the network.**
  - c. **High-availability, multi-machine cluster with an external database**
  - d. *Active/passive redundancy multi-machine with an external database*
4. Which of the following two features are not provided by a basic Ansible Tower license? (Choose two.)
  - a. *Tower dashboard*
  - b. *Job scheduling*
  - c. **System tracking**
  - d. *Role-based access control*
  - e. **Logging aggregation**

# Installing Ansible Tower

## Objectives

After completing this section, students should be able to install a new Ansible Tower on a single node using the `setup.sh` script.

## Installation Requirements

Ansible Tower can be installed and is supported on 64-bit x86\_64 versions of Red Hat Enterprise Linux 7, CentOS 7, Ubuntu 14.04 LTS, and Ubuntu 16.04 LTS. The following are the requirements for installing Ansible Tower on a Red Hat Enterprise Linux 7 system. Details may vary slightly for other supported operating systems.

### Operating System

For Red Hat Enterprise Linux installations, the Ansible Tower 3.1 server is supported on systems running Red Hat Enterprise Linux 7.2 or later on the 64-bit x86\_64 processor architecture.

### Web Browser

A currently supported version of Mozilla Firefox or Google Chrome is required for connecting to the Ansible Tower web interface. Other HTML5-compliant web browsers may work but are not fully tested or supported.

### Memory

A minimum of 2 GB of RAM is required on the Tower host. 4 GB or more is recommended.

The actual memory requirement is dependent upon the maximum number of hosts Ansible Tower is expected to configure in parallel. This is managed by the `forks` configuration parameter in the job template or system configuration. Red Hat recommends that 4 GB of memory be available for each 100 forks. A deeper discussion on Ansible Tower memory requirements and how they are influenced by job concurrency settings can be found in the *Ansible Tower User Guide*, in the "Job Concurrency" section of the "Jobs" chapter, at <http://docs.ansible.com>.

### Disk Storage

At least 20 GB of hard disk space is required for Ansible Tower. In order for the Ansible Tower installation to succeed, 10 GB of this space must be available for the `/var` directory.

This minimum disk storage requirement does not include storage required for Tower's System Tracking feature. If this feature is enabled, calculate additional storage requirement based on the number of hosts tracked, scans performed, and the amount of data collected. A formula to estimate this is available in the "Requirements" chapter of the *Ansible Tower Installation and Reference Guide*.

### Ansible

Installation of Ansible Tower is performed by executing a shell script that runs an Ansible playbook. Older versions of Ansible Tower required that the latest stable version of Ansible was installed prior to installation, but the current installation program will automatically attempt to install Ansible and its dependencies if they are not already present.

Red Hat Enterprise Linux 7 users must enable the `extras` repository.

If the RPM package for Ansible (or DEB for Ubuntu) is already installed on the system, the Tower installer will not attempt to reinstall it. If you have installed Ansible on the server using the

Python package manager **pip**, the Tower installer *will* attempt to reinstall Ansible. For Tower to work correctly, the latest stable version of Ansible should be installed using your distribution's package manager.

#### SELinux

Ansible Tower supports the **targeted** SELinux policy, which can be set to enforcing mode, permissive, or disabled. Other SELinux policies are not supported.

#### Managed clients

The installation requirements above apply to the Ansible Tower server, not to the machines it manages with Ansible. Those systems should meet the usual requirements for machines managed with the version of Ansible installed on the Ansible Tower server.

## Ansible Tower Licensing and Support

Administrators interested in evaluating Ansible Tower can obtain a trial license at no cost. Instructions on how to get started are available at <https://www.ansible.com/tower-trial>.

Administrators interested in progressing beyond trial licensing can choose from three types of Tower subscriptions:

- Self-Support

Targeted at small deployments, this includes a basic Tower subscription, with maintenance and upgrades of the software but no technical support or service level agreement (SLA). Some "enterprise" features of Tower are not included. Versions supporting up to 250 managed nodes are available, larger deployments should look at the Enterprise subscriptions.

- Enterprise: Standard

The Enterprise: Standard edition includes an enterprise Tower subscription with entitlement to all Tower features and 8x5 technical support. Pricing is based on the number of nodes managed. Ansible Playbook support is included, which at the time of writing includes help with runtime execution problems in Tower, assistance with errors and tracebacks, and limited recommended practice guidance on the current or previous minor release of Ansible.

- Enterprise: Premium

The Enterprise: Premium edition also includes an enterprise Tower subscription with software maintenance and upgrades and all Tower features, but with entitlement to 24x7 technical support. Pricing is based on the number of nodes managed, and Ansible Playbook support is included.

Detailed information on Ansible Tower pricing and support is available at <http://www.ansible.com>.

#### Licenses for Tower Components

Tower makes use of various software components, some of which may be open source. Licenses for each of these specific components are provided under the **/usr/share/doc/ansible-tower** directory.

## Ansible Tower Installers

Two different installation packages are available for Ansible Tower.

The standard **setup** Ansible Tower installation program can be downloaded from <http://releases.ansible.com/ansible-tower/setup/>. The latest version of Ansible Tower for

RHEL 7 is always located at <http://releases.ansible.com/ansible-tower/setup/ansible-tower-setup-latest.el7.tar.gz>. This archive is smaller, but requires Internet connectivity to download Ansible Tower packages from various package repositories.

A different, bundled installer for RHEL 7 is available at <http://releases.ansible.com/ansible-tower/setup-bundle/ansible-tower-setup-bundle-latest.el7.tar.gz>. This archive includes an initial set of RPM packages for Ansible Tower so that it may be installed on systems disconnected from the Internet. Those systems still need to be able to get software packages for Red Hat Enterprise Linux 7 and the RHEL 7 Extras channel from reachable sources. This may be preferred by administrators in higher security environments. This installation method is not currently available for Ubuntu.

## Installing Tower

The following is the procedure for using the bundled installer to install Ansible Tower on a single RHEL 7.3 system with access to the Red Hat Enterprise Linux 7 Extras repository. The exercise after this section will go over this in more detail.

1. As the **root** user, download the Ansible Tower setup bundle to the system.
2. Extract the Tower setup bundle and change into the directory containing the extracted contents.

```
[root@towerhost ~]# tar xzf ansible-tower-setup-bundle-3.1.1-1.el7.tar.gz
[root@towerhost ~]# cd ansible-tower-setup-bundle-3.1.1-1.el7
```

3. In that directory, the **inventory** file needs to be edited in order to set passwords for the Ansible Tower **admin** account (**admin\_password**), the PostgreSQL database user account (**pg\_password**), and the RabbitMQ messaging user account (**rabbitmq\_password**).

```
[tower]
localhost ansible_connection=local

[database]

[all:vars]
admin_password='myadminpassword'

pg_host=''
pg_port=''

pg_database='awx'
pg_username='awx'
pg_password='somedatabasepassword'

rabbitmq_port=5672
rabbitmq_vhost=tower
rabbitmq_username=tower
rabbitmq_password='and-a-messaging-password'
rabbitmq_cookie=cookiemonster

Needs to be true for fqdns and ip addresses
rabbitmq_use_long_name=false
```



## Warning

These passwords can be anything and should be set to something secure. The **admin** user has full control of the Tower server, and the ports for PostgreSQL and the RabbitMQ service are exposed to external hosts by default.

- Run the Ansible Tower installer by executing the **setup.sh** script.

```
[root@towerhost ansible-tower-setup-bundle-3.1.1-1.el7]# ./setup.sh
[warn] Will install bundled Ansible
Loaded plugins: langpacks, search-disabled-repos
Examining bundle/repos/epel/ansible-2.2.1.0-1.el7.noarch.rpm:
 ansible-2.2.1.0-1.el7.noarch
Marking bundle/repos/epel/ansible-2.2.1.0-1.el7.noarch.rpm to be installed
... Output omitted ...
The setup process completed successfully.
Setup log saved to /var/log/tower/setup-2017-02-27-10:52:44.log
```

- Once the installer has completed successfully, connect to the Ansible Tower system using a web browser. You should be redirected to an HTTPS login page.

The web browser may generate a warning regarding a self-signed HTTPS certificate presented by the Ansible Tower website. Replacing the default self-signed HTTPS certificate for the Ansible Tower web interface with a properly CA-signed certificate is discussed later in this course.

- Log in to the Ansible Tower web interface as the Tower administrator using the **admin** account and the password you set in the installer's **inventory** file.
- Once successfully logged into the Tower web interface for the first time, you are prompted to enter a license and accept the end user license agreement. Enter the Ansible Tower license provided by Red Hat and accept the end user license agreement.

At this point, the **admin** user is presented with the Tower dashboard. The next section provides an orientation to the Tower interface in more detail.



## References

- Ansible Tower Quick Installation Guide* for Ansible Tower 3.1.1  
| <http://docs.ansible.com/ansible-tower/3.1.1/html/quickinstall/>
- Ansible Tower Installation and Reference Guide* for Ansible Tower 3.1.1  
| <http://docs.ansible.com/ansible-tower/3.1.1/html/installandreference/>
- Ansible Tower User Guide* for Ansible Tower 3.1.1  
| <http://docs.ansible.com/ansible-tower/3.1.1/html/userguide/>
- Ansible Tower Administration Guide* for Ansible Tower 3.1.1  
| <http://docs.ansible.com/ansible-tower/3.1.1/html/administration/>

# Guided Exercise: Installing Ansible Tower

In this exercise, you will install a single-node Ansible Tower instance on **tower**.

## Outcomes

You should be able to successfully install Ansible Tower and its license, resulting in a running Ansible Tower server.

## Before you begin

Run **lab tower-install setup** on **workstation** to configure **tower.lab.example.com** with a Yum repository containing package dependencies from the Red Hat Enterprise Linux and Extras repositories required for Ansible Tower installation.

```
[student@workstation ~]$ lab tower-install setup
```

## Steps

1. Download the Ansible Tower setup bundle to the **tower** system.

- 1.1. Log in to the **tower** system as the **root** user.

```
[student@workstation ~]$ ssh root@tower
```

- 1.2. Download the Ansible Tower setup bundle using the **curl** command.

```
[root@tower ~]# curl -O -J http://content.example.com/ansible2.3/x86_64/dvd/ansible-tower/ansible-tower-setup-bundle-3.1.1-1.el7.tar.gz
```

2. Extract the Tower setup bundle. Change into the directory containing the extracted contents.

```
[root@tower ~]# tar xzf ansible-tower-setup-bundle-3.1.1-1.el7.tar.gz
[root@tower ~]# cd ansible-tower-setup-bundle-3.1.1-1.el7
```

3. Set the passwords for the Ansible Tower administrator account, database user account, and messaging user account to **redhat** by modifying their respective entries in the **inventory** file used by the Tower installer playbook.

```
[root@tower ansible-tower-setup-bundle-3.1.1-1.el7]# grep password inventory
admin_password='redhat'
pg_password='redhat'
rabbitmq_password='redhat'
```

4. Run the Ansible Tower installer by executing the **setup.sh** script. The script may take up to 30 minutes to complete. Ignore the errors in the script output as they are related to verification checks performed by the installer playbook.

```
[root@tower ansible-tower-setup-bundle-3.1.1-1.el7]# ./setup.sh
[warn] Will install bundled Ansible
Loaded plugins: langpacks, search-disabled-repos
```

```
Examining bundle/repos/epel/ansible-2.2.1.0-1.el7.noarch.rpm:
 ansible-2.2.1.0-1.el7.noarch
Marking bundle/repos/epel/ansible-2.2.1.0-1.el7.noarch.rpm to be installed
... Output omitted ...
The setup process completed successfully.
Setup log saved to /var/log/tower/setup-2017-02-27-10:52:44.log
```

- Once the installer has completed successfully, exit the console session on the **tower** system.

```
[root@tower ansible-tower-setup-bundle-3.1.1-1.el7]# exit
```

- Launch the Firefox web browser from **workstation** and connect to your Ansible Tower at **https://tower.lab.example.com**. Firefox warns you that the Ansible Tower server's security certificate is not secure. Add and confirm the security exception for the self-signed certificate.
- Log in to the Tower web interface as the Tower administrator using the **admin** account and the **redhat** password.
- Once you have successfully logged in to the Tower web interface for the first time, you are prompted to enter a license and accept the end user license agreement.

Upload the Ansible Tower license and accept the end user license agreement.

- On **workstation**, download the Ansible Tower license provided at **http://materials.example.com/Ansible-Tower-license.txt**.

- In the Tower web interface, click **BROWSE** and then select the license file downloaded earlier.

- Select the checkbox next to **I agree to the End User License Agreement** to indicate acceptance.

- Click **SUBMIT** to submit the license and accept the license agreement.

# Navigating the Ansible Tower Web Interface

## Objectives

After completing this section, students should be able to navigate and describe the Ansible Tower web user interface, and successfully launch a job using the demo job template, project, credential, and inventory.

## Using Ansible Tower

This section provides an overview of how to use the Ansible Tower web interface to launch a job with an example Ansible playbook, an inventory, and some access credentials for the machines in the inventory. Along the way, it provides an orientation to the web interface itself.

The basic idea is that Tower is configured with a number of Ansible *projects* that contain playbooks. It is also configured with a number of Ansible *inventories* and the necessary machine *credentials* to log in to inventory hosts and escalate privileges. A *job template* is set up by an administrator which specifies which playbook from which project should be run on the hosts in a particular inventory using particular machine credentials. A *job* happens when a user runs a playbook on an inventory by launching a job template.

## Tower Dashboard

Upon successful login to the Tower web interface, users are presented with the Tower Dashboard, the main control center for Ansible Tower.

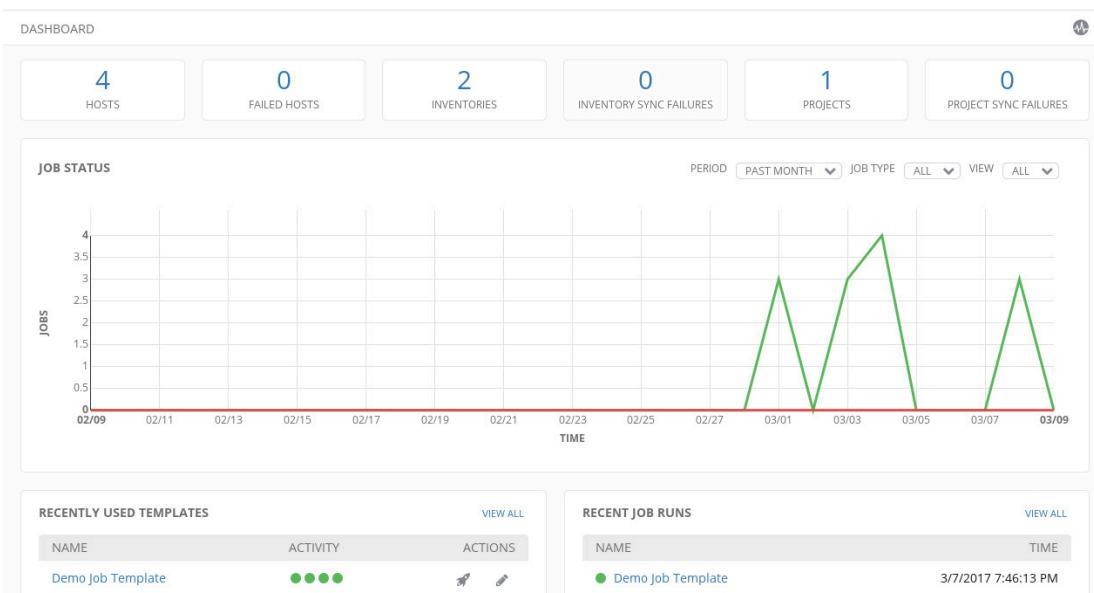


Figure 11.2: Tower Dashboard

This dashboard screen is composed of four reporting sections:

### Summary

Across the top of the dashboard is a summary report of the status of managed hosts, inventories, and Ansible projects. Clicking on a cell in the summary section takes the user to the detailed dashboard screen for the reported metric.

### Job Status

A *job* is an attempted run of a playbook by Ansible Tower. This section provides a graphical display of the number of successful and failed jobs over time. This graph can be adjusted in several ways:

- The **PERIOD** dropdown menu can be used to change the time window for the plotted graph between either the most recent day, week, or month.
- The **JOB TYPE** dropdown menu can be used to select which job types to include on the graph.
- The **VIEW** drop-down menu can be used to choose between graphing all job status, only failed job, or only successful jobs.

### Recently Used Templates

This section reports a listing of job templates which was recently used for the execution of jobs.

- For each job template used, the results of each associated job run is indicated under the **ACTIVITY** column by a colored dot, with green indicating success and red indicating failure.
- Under the **ACTIONS** column are controls for the use and modification of the job template.
- Clicking the **VIEW ALL** link displays all job templates, not just the ones which have recently been used for job execution.

### Recent Job Runs

This section provides a listing of recently executed jobs along with their date and time of execution. Each job run listed is preceded by a colored dot which represents the outcome of the run. A green dot represents a successful run while a red dot represents a failed run.

## Quick Navigation Links

In the upper left portion of the Tower web interface is a collection of navigation links to commonly used Tower resources. The Ansible Tower icon links users back to the Tower Dashboard page. The Projects, Inventories, Templates, and Jobs links direct to the administrative screen for each of these four Tower resources.



Figure 11.3: Quick navigation links

- *Projects*: In Tower, a project represents a collection of related Ansible playbooks.
- *Inventories*: As in Ansible, inventories in Ansible Tower contain a collection of hosts to be managed.
- *Templates*: The template resource defines the parameters which are to be used for the execution of an Ansible playbook by Ansible Tower.
- *Jobs*: A job represents Tower's execution of an Ansible playbook against an inventory of hosts.

## Administration Tool Links

Across the upper-right portion of the Tower web interface are links to various Tower administration tools.



Figure 11.4: Administration tool links

### Account configuration

The current user account name is displayed as a link. Clicking the link directs users to their account configuration page, where they can modify their username, password, or account user type.

### Settings

The gear icon links users to the **Settings** menu. This menu provides access to administrative interfaces for various Tower resources.

#### My View

To the right of the **Settings** icon is a report icon, which links to Tower's **My View** screen. This screen reports on the same information as the Tower Dashboard but only for the jobs executed by the logged in user.

#### View Documentation

The book icon to the right of the **My View** represents the **View Documentation** link. Clicking on this link opens the online Ansible Tower documentation web site in a new window.

#### Log Out

The power icon link to the right of the **View Documentation** icon is used to log out of the Tower web interface.

## Tower Settings

As previously mentioned Tower's Settings menu can be accessed by clicking the Settings icon in the administration toolbar. The **Settings** menu provides access to the following management interfaces for Tower resources which are not accessible through the quick navigation links.

The screenshot shows the 'SETTINGS' section of the Tower interface. At the top, there are links for TOWER, PROJECTS, INVENTORIES, TEMPLATES, and JOBS. On the far right, there is a user icon labeled 'admin' and a 'Settings' icon. Below these are two rows of management interfaces:

- ORGANIZATIONS**: Group all of your content to manage permissions across departments in your company.
- USERS**: Allow others to sign into Tower and own the content they create.
- TEAMS**: Split up your organization to associate content and control permissions for groups.
- CREDENTIALS**: Add passwords, SSH keys, etc. for Tower to use when launching jobs against machines, or when syncing inventories or projects.

- MANAGEMENT JOBS**: Manage the cleanup of old job history, activity streams, data marked for deletion, and system tracking info.
- INVENTORY SCRIPTS**: Create and edit scripts to dynamically load hosts from any source.
- NOTIFICATIONS**: Create templates for sending notifications with Email, HipChat, Slack, and SMS.
- VIEW YOUR LICENSE**: View and edit your license information.

- CONFIGURE TOWER**: Edit Tower's configuration.
- ABOUT TOWER**: View information about this version of Ansible Tower.

Figure 11.5: Settings

### Organizations

This interface is used for managing organization entities within Tower. An organization represents a logical collection of other Tower resources, such as Teams, Projects, and Inventories.

Organizations are often used for departmental separation within an enterprise. An organization is the highest level at which Tower's role-based access control system can be applied.

### **Users**

This interface allows for Tower user management. Users are granted access to Tower and then assigned roles which determine their access to Tower resources.

### **Teams**

Tower Teams can be administered through this interface. Teams represent a group of Users. Like Users, teams can also be assigned roles for access to Tower resources.

### **Credentials**

The management of Credentials is performed through this interface. Credentials are authentication data used by Tower to do such things as logging in to managed hosts to run plays, synchronizing inventory data from external sources, and downloading updated project materials from version control systems.

### **Management Jobs**

This interface is used to administer system jobs which perform cleanup of metrics and activity information stored by Tower operations.

### **Inventory Scripts**

This interface manages scripts used for the generation and update of dynamic inventories from external sources, such as cloud providers and configuration management databases (CMDBs)

### **Notifications**

Administration of Notification Templates is conducted through this interface. These templates define the set of configuration parameters needed to generate notifications using a variety of message delivery tools, such as email, IRC, and HipChat.

### **View Your License**

This interface provides details of the installed license and can also be used to perform administrative licensing tasks such as license installation and upgrade.

### **Configure Tower**

Configurable options related to authentication, job execution, and the Tower web interface are performed through this interface.

### **About Tower**

Information regarding the version of Tower installed can be displayed using this link.

## **General Controls**

In addition to the navigational and administrative controls previously outlined, some additional controls are used throughout the Tower web interface.

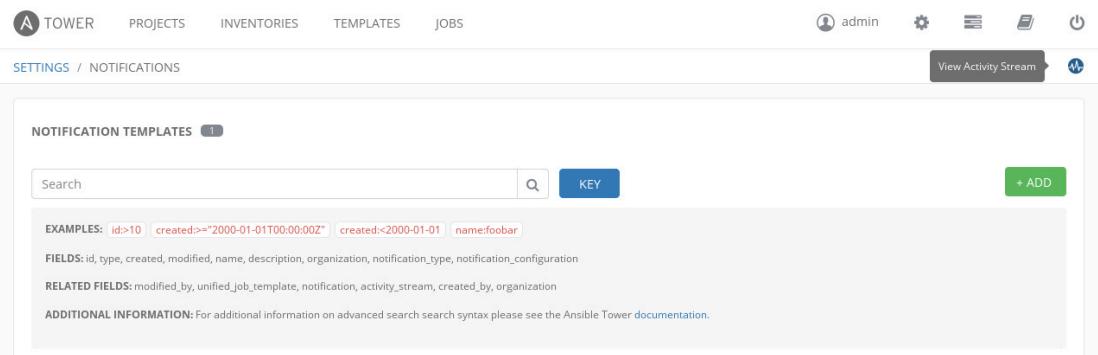


Figure 11.6: Key

### Breadcrumb Navigation Links

As a user navigates through the Tower web interface, a set of breadcrumb navigation links is created just below the quick navigation links in the upper left corner of the screen. This series of links not only make clear the path which leads to each screen but also provides a quick way to return to a previous screens in the navigated path.

### Activity Streams

On most screens in the Tower web interface, there is a **View Activity Stream** link under the Administrative Tools links at the top right portion. This link can be used to produce a page reporting activities which have occurred and are related to the screen that the button is clicked on. For example, when the Activity Streams button is clicked on the Projects screen, the information regarding the time, executor, and the nature of project-related activities is displayed.

### Search Fields

Throughout the Tower web interface are search fields which can be used to search or filter through data sets. Each search field accepts search criteria which can be used to narrow down the search result.

### Key

A guide detailing the correct syntax of the specific criteria for each search field can be displayed by clicking the **Key** button located next to the search field.

The **Key** button is also used to define the options provided by other input fields within Tower such as dropdown lists.

## Configuration of a Job Template

When installed, Ansible Tower is preconfigured with a demonstration Job Template which can be used as an example to see how a Job Template is constructed and to test the operation of the Tower. The following discussion looks at the components that make up this example. The exercise that follows this section will provide a more detailed hands-on walk through.

- Under **INVENTORIES**, a **Demo Inventory** has been configured. This is a static inventory with no hostgroups and one host, **localhost**. Clicking on that host in the inventory reveals that it has the inventory variable **ansible\_connection: local** set.

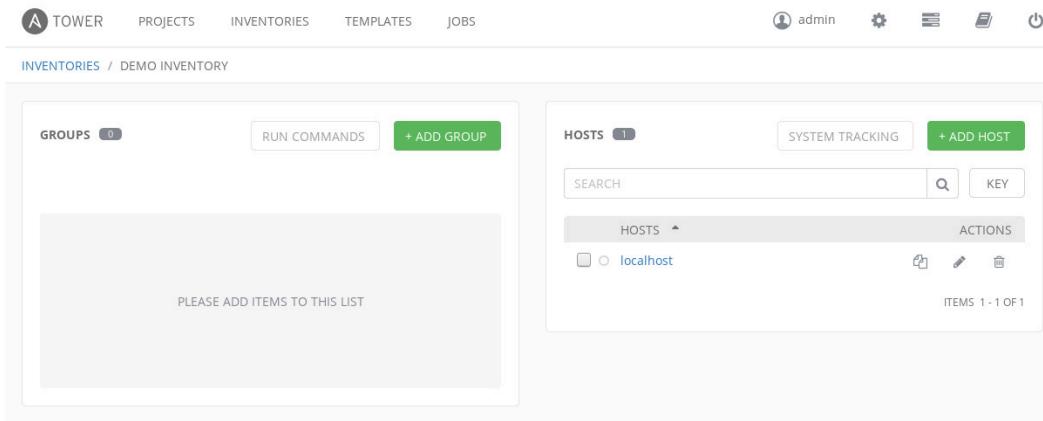


Figure 11.7: Demo Inventory

2. Under Settings (the gear icon), there is a **CREDENTIALS** link. A machine credential named **Demo Credential** has been created which contains information that could be used to authenticate access to machines in an inventory.
3. Under **PROJECTS**, a **Demo Project** has been configured. This Project is configured to get Ansible project materials, including playbooks, from a local directory on the Tower system.

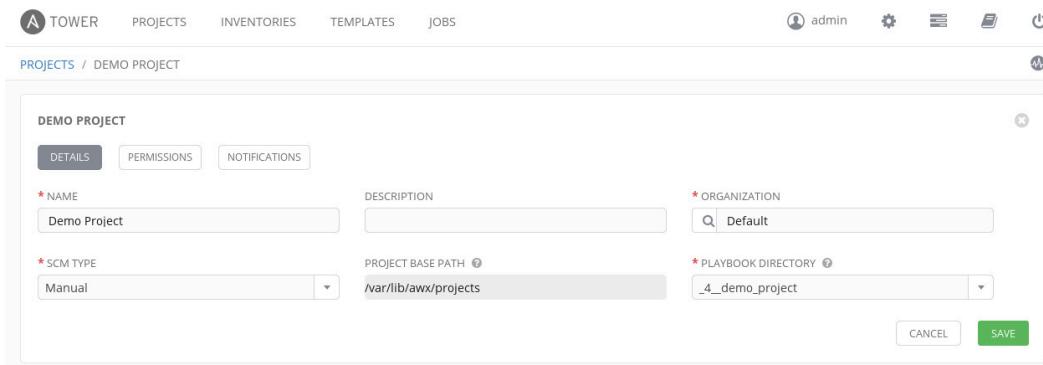


Figure 11.8: Demo Project

4. Under **TEMPLATES**, a **Demo Job Template** has been configured. This Job Template is configured as a normal playbook run (Job Type is Run), using the **hello\_world.yml** playbook from **Demo Project**.

It runs on the machines in the **Demo Inventory**, using the **Demo Credential** to authenticate to those machines. Privilege escalation will not be enabled (**hello\_world.yml** doesn't need it). Were it needed, the **Demo Credential** would need to provide the necessary information. (In fact, because the Job Template is not using privilege escalation and is running only on machines using **ansible\_connection: local**, there is very little information needed in the **Demo Credential**.)

No extra variables are set (analogous to the **-e** or **--extra-vars** option of an **ansible-playbook** command).

The screenshot shows the 'DEMO JOB TEMPLATE' configuration dialog. At the top, there are tabs for DETAILS, COMPLETED JOBS, PERMISSIONS, NOTIFICATIONS, and ADD SURVEY. The ADD SURVEY tab is currently selected.

**Job Details:**

- NAME:** Demo Job Template
- DESCRIPTION:** (empty)
- JOB TYPE:** Run (selected from a dropdown menu)
- Prompt on launch:** (checkbox)

**Inventory and Project:**

- INVENTORY:** Demo Inventory
- PROJECT:** Demo Project
- PLAYBOOK:** hello\_world.yml
- Prompt on launch:** (checkbox)

**Machine and Cloud Credentials:**

- MACHINE CREDENTIAL:** Demo Credential
- CLOUD CREDENTIAL:** (empty)
- NETWORK CREDENTIAL:** (empty)
- Prompt on launch:** (checkbox)

**Execution Parameters:**

- FORKS:** 0
- LIMIT:** (empty)
- VERBOSITY:** 0 (Normal)
- Prompt on launch:** (checkbox)

**Tags and Options:**

- JOB TAGS:** (empty)
- SKIP TAGS:** (empty)
- OPTIONS:**
  - Enable Privilege Escalation (checkbox)
  - Allow Provisioning Callbacks (checkbox)
  - Enable Concurrent Jobs (checkbox)
- Prompt on launch:** (checkbox)

**Labels:** (empty)

**Extra Variables:** (radio buttons) YAML (selected) JSON

**EXTRA VARIABLES:** (text area containing '1')  
1

**Prompt on launch:** (checkbox)

**Buttons:** CANCEL (grayed out), SAVE (green)

Figure 11.9: Demo Job Template

## Launching a Job

Once a Job Template is configured, it can be used to launch Jobs repeatedly using the same parameters. A Job Template is somewhat like a canned **ansible-playbook** command complete with options and arguments that's been written down or is in a shell history. When the Job Template is used to launch a Job, it's like running that **ansible-playbook** command from a shell prompt.

The following discussion looks at what happens when the **Demo Job Template** is used to launch a Job. The exercise following this section also covers this in a more detailed hands-on manner.

- Under **TEMPLATES**, the **Demo Job Template** should be listed. Across from the name of the Job Template is a rocket icon ("Start a job using this template"). Clicking that icon launches the job using the settings in the Job Template.

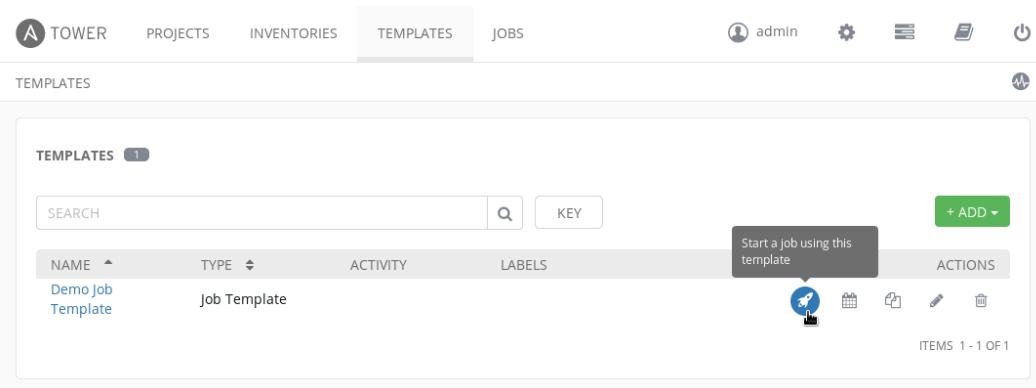


Figure 11.10: Launching a Job

2. As the Job runs, a new status page opens that provides real-time information about the progress of the Job. The **DETAILS** pane provides basic information about the job being run: when it was launched, who launched it, what Job Template, Project, and Inventory were used, and so on. While running, the job status is **Pending**.
3. As the Job executes, the status page also includes the output from the job run in a job output pane. The job run output resembles the output generated when the **ansible-playbook** command is executed on an Ansible playbook. This can be used for troubleshooting purposes. In the example screenshot that follows, the play and tasks in the playbook ran successfully.

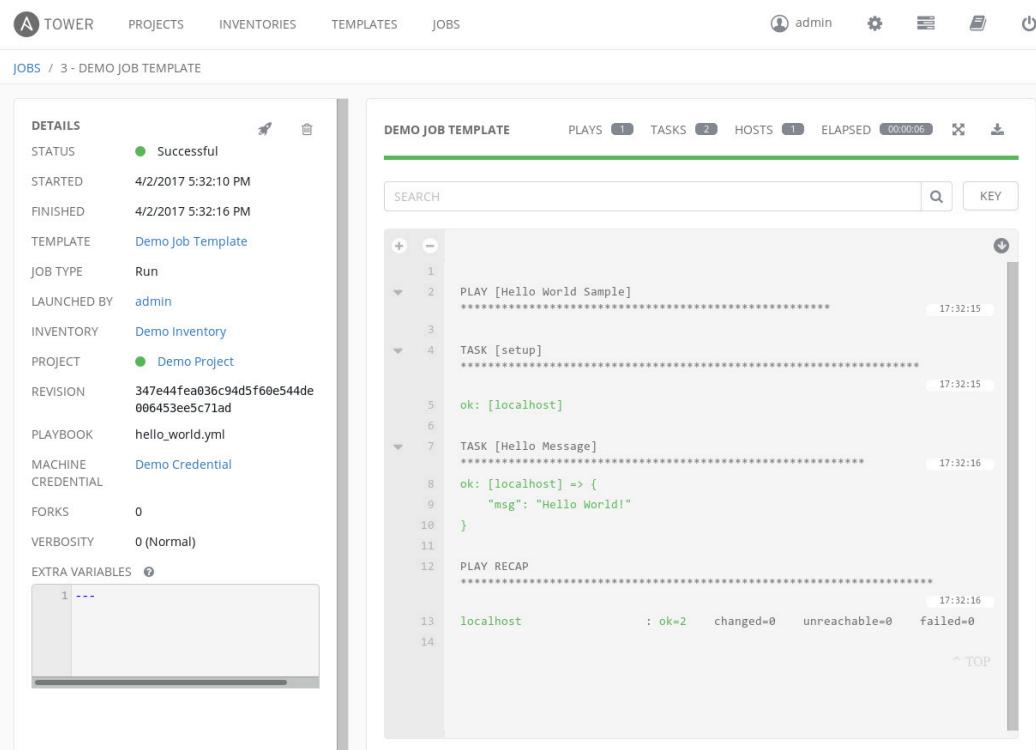


Figure 11.11: Example Job output

4. After the Job completes, the Tower Dashboard (reachable by clicking on the **TOWER** link at the upper left corner of the web interface) has a link to the status page for the Job run under **RECENT JOB RUNS**. The other statistics on the Tower Dashboard are also updated.
5. Under **JOBS**, all the Jobs that have run on the Tower are listed. Clicking on the name of the Job listed on this page also displays the status page logged for that Job.

The screenshot shows the 'JOBS' section of the Ansible Tower interface. At the top, there are tabs for 'JOBS' (which is selected), 'PROJECTS', 'INVENTORIES', 'TEMPLATES', and 'JOBS'. On the right side, there are user icons for 'admin' and various system settings. Below the tabs, the word 'JOBS' is centered above a table. The table has columns: ID, NAME, TYPE, FINISHED, and ACTIONS. There are four rows of data:

ID	NAME	TYPE	FINISHED	ACTIONS
3	Demo Job Template	Playbook Run	4/2/2017 5:32:16 PM	
4	Demo Project	SCM Update	4/2/2017 5:32:09 PM	
2	Demo Project	SCM Update	4/2/2017 4:34:28 PM	
1	Cleanup Fact Details	Management Job	4/1/2017 6:31:14 PM	

At the bottom right of the table area, it says 'ITEMS 1 - 4 OF 4'.

Figure 11.12: Example JOBS screen

## References

*Ansible Tower Quick Setup Guide for Ansible Tower 3.1.1*  
 | <http://docs.ansible.com/ansible-tower/3.1.1/html/quickstart/>

*Ansible Tower User Guide for Ansible Tower 3.1.1*  
 | <http://docs.ansible.com/ansible-tower/3.1.1/html/userguide/>

# Guided Exercise: Navigating the Ansible Tower Web Interface

In this exercise, you will navigate through the Ansible Tower web interface and launch a job.

## Outcomes

You should be able to browse through and interact with the Project, Inventory, Credential, and Template screens in the Ansible Tower web interface to successfully launch a job.

## Before you begin

You should have an Ansible Tower instance installed and configured from an earlier exercise running on the **tower** system.

Log in as the **student** user on **workstation** and run **lab tower-webui setup**. This setup script configures the **tower** virtual machine for the exercise.

```
[student@workstation ~]$ lab tower-webui setup
```

## Steps

1. Log in to the Ansible Tower web interface running on the **tower** system using the **admin** account and the **redhat** password.
2. Identify the Project that was created during the Ansible Tower installation and determine its source.
  - 2.1. Click **PROJECTS** in the top navigation menu to display the list of Projects. You should see a Project named **Demo Project**, which was created during the Ansible Tower installation.
  - 2.2. Click the **Demo Project** link to view the details of the Project.
  - 2.3. Look at the value of the **SCM TYPE**, **PROJECT BASE PATH**, **PLAYBOOK DIRECTORY** fields to determine the origin of the Project. You should see that **Demo Project** was obtained from the directory, **\_4\_demo\_project**, which resides under the **/var/lib/awx/projects** local directory on the Tower system.
3. Browse the Inventory, which was created during the Ansible Tower installation, and determine its managed hosts.
  - 3.1. Click **INVENTORIES** in the top navigation menu to display the list of known Inventories. You should see an Inventory named **Demo Inventory**, which was created during the Ansible Tower installation.
  - 3.2. Click the **Demo Inventory** link to view the details of the Inventory. Under the **HOSTS** section, you should see that the Inventory is composed of just one host, **localhost**.
4. View the details of the Credential, which was created during the Ansible Tower installation.
  - 4.1. Click the **Settings** icon in the administration toolbar to display the list of administrative interfaces.
  - 4.2. Click **CREDENTIALS** to view the list of Credentials.

- 
- 4.3. Click the **Demo Credential** link to view the details of the Credential. You should see that the Credential is a machine credential that uses the username **admin**.
  5. Identify the Job Template, which was created during the Ansible Tower installation. Click **TEMPLATES** in the top navigation menu to display the list of existing Job Templates. You should see a Job Template named **Demo Job Template**, which was created during the Ansible Tower installation.
  6. Determine the Inventory, Project, and Credential utilized by the **Demo Job Template** Job Template.
    - 6.1. Click the **Demo Job Template** link to view the details of the Job Template.
    - 6.2. Look at the **INVENTORY** field. You should see that the Job Template uses the **Demo Inventory** Inventory.
    - 6.3. Look at the **PROJECT** field. You should see that the Job Template uses the **Demo Project** Project.
    - 6.4. Look at the **PLAYBOOK** field. You should see that the Job Template executes a **hello\_world.yml** playbook.
    - 6.5. Look at the **MACHINE CREDENTIAL** field. You should see that the Job Template uses the **Demo Credential** Credential.
  7. Launch a job using the **Demo Job Template** Job Template.
    - 7.1. Exit the template details view by clicking the **TEMPLATES** link in the breadcrumb navigation menu near the top of the screen.
    - 7.2. On the **TEMPLATES** screen, click the rocket icon under the **ACTIONS** column of the **Demo Job Template** row. This launches a job using the parameters configured in the **Demo Job Template** template and redirects you to the job details screen. As the job executes, the details of the job execution as well as its output is displayed.
  8. Determine the outcome of the job execution.
    - 8.1. When the job has completed successfully, the **STATUS** value changes to **Successful**.
    - 8.2. Review the output of the job execution to determine, which tasks were executed. You should see that the **msg** module was used to successfully display a "**Hello World!**" message
  9. Review the changes to the dashboard reflecting the job execution.
    - 9.1. Click **TOWER** in the upper-left corner of the screen to return to the dashboard.
    - 9.2. Review the **JOB STATUS** graph. The green line on the graph indicates the number of recent successful job executions.
    - 9.3. Review the **RECENT JOB RUNS** section. This section provides a list of the jobs recently executed as well as their results. The **Demo Job Template** entry indicates that the

Job Template was used to execute a job. The green dot at the beginning of the entry indicates the successful completion of the executed job.

# Quiz: Implementing Ansible Tower

Choose the correct answer(s) to the following questions:

1. Which three of the following statements are a component of Ansible Tower 3.1 architecture? (Choose three).
  - a. Django web application
  - b. MongoDB database
  - c. PostgreSQL database
  - d. RabbitMQ messaging system
2. Which three of the following statistics are reported by the Tower Dashboard? (Choose three).
  - a. Number of jobs executed
  - b. Number of credentials stored
  - c. Status of executed jobs
  - d. Number of hosts managed
3. Which of the following is not a requirement for Tower 3.1 installation?
  - a. An existing installation of Ansible
  - b. SELinux "targeted" policy
  - c. Minimum of 2GB of RAM
  - d. Minimum of 20GB of hard disk space
4. Which two of the following statements regarding the bundled Tower installer are incorrect? (Choose two).
  - a. Includes additional software dependencies needed by Tower.
  - b. Requires access to third party repositories in order to meet software dependencies.
  - c. Requires access to Red Hat Enterprise Linux 7 Extras repository for software dependencies.
  - d. Supports installations on systems running Ubuntu
5. Which of the following Tower resources cannot be accessed through the quick navigation links?
  - a. Projects
  - b. Inventories
  - c. Notifications
  - d. Templates

## Solution

Choose the correct answer(s) to the following questions:

1. Which three of the following statements are a component of Ansible Tower 3.1 architecture? (Choose three).
  - a. **Django web application**
  - b. **MongoDB database**
  - c. **PostgreSQL database**
  - d. **RabbitMQ messaging system**
2. Which three of the following statistics are reported by the Tower Dashboard? (Choose three).
  - a. **Number of jobs executed**
  - b. **Number of credentials stored**
  - c. **Status of executed jobs**
  - d. **Number of hosts managed**
3. Which of the following is not a requirement for Tower 3.1 installation?
  - a. **An existing installation of Ansible**
  - b. **SELinux "targeted" policy**
  - c. **Minimum of 2GB of RAM**
  - d. **Minimum of 20GB of hard disk space**
4. Which two of the following statements regarding the bundled Tower installer are incorrect? (Choose two).
  - a. Includes additional software dependencies needed by Tower.
  - b. **Requires access to third party repositories in order to meet software dependencies.**
  - c. Requires access to Red Hat Enterprise Linux 7 Extras repository for software dependencies.
  - d. **Supports installations on systems running Ubuntu**
5. Which of the following Tower resources cannot be accessed through the quick navigation links?
  - a. Projects
  - b. Inventories
  - c. **Notifications**
  - d. Templates

# Summary

In this chapter, you learned:

- Ansible Tower 3.1 is a centralized management solution for Ansible projects.
- Ansible Tower is offered with two installer options. The standard installer downloads packages from network repositories. The bundled installer includes third party software dependencies.
- Job Templates are prepared commands to execute Ansible playbooks. Important components of a job template include an inventory, machine credential, Ansible project and playbook.
- A Job is launched from a Job Template, and represents a single run of a playbook on an inventory of machines.
- Tower Dashboard provides summaries on the status of hosts, inventories, projects, and executed jobs.
- Quick navigation links at the top of the Tower web interface provide access to commonly used Tower resources.
- The **Settings** menu in the Tower web interface provides access to the management interfaces of Tower resources which are not accessible through the quick navigation links.





## CHAPTER 12

# IMPLEMENTING ANSIBLE IN A DEVOPS ENVIRONMENT

Overview	
<b>Goal</b>	Implement Ansible in a DevOps environment using Vagrant.
<b>Objectives</b>	<ul style="list-style-type: none"><li>• Describe Ansible in a DevOps environment and provision Vagrant machines.</li><li>• Deploy Vagrant in a DevOps environment.</li></ul>
<b>Sections</b>	<ul style="list-style-type: none"><li>• Provisioning Vagrant Machines (and Guided Exercise)</li><li>• Deploying Vagrant in a DevOps Environment (and Guided Exercise)</li></ul>
<b>Lab</b>	<ul style="list-style-type: none"><li>• Implementing Ansible in a DevOps Environment</li></ul>

# Provisioning Vagrant Machines

## Objectives

After completing this section, students should be able to:

- Discuss Ansible in a DevOps environment.
- Provision a Vagrant machine.

## DevOps in the enterprise

In recent years, the DevOps approach has seen increasing adoption in the enterprise as organizations strive to resolve the conflict between their development and operational teams. DevOps derives its name from the core principle that software development and operations performance can be improved and accelerated through better communication, integration, and cooperation between software developers and IT operations professionals.

### Infrastructure as code

DevOps places a strong focus on the ability to build and maintain essential components with automated, programmatic procedures. One key DevOps concept is the idea of *Infrastructure as Code (IaC)*. This concept is an important and groundbreaking paradigm shift for the many system administrators who currently manage their infrastructure through manual execution of administrative commands and editing of configuration files. By designing, implementing, and managing infrastructure as code, configurations can be predictably and consistently deployed and replicated throughout an environment.

In recent years, the Ansible software has become a popular tool for managing infrastructure code. Ansible allows tasks such as software installations and server configurations to be automated. This automation removes the introduction of human errors resulting from manual administration. It also allows for the repeatability of routine tasks and removes the complexity of these tasks so they can be easily performed even by entry level administrators. Ansible's use can greatly improve an organization's operational efficiency and simultaneously enhance the predictability of successful outcomes.

Ansible's architecture allows for the centralization of configuration changes. This design results in greater control over the infrastructure code by effectively implementing a single point of entry for changes to the infrastructure.

Ansible's declarative nature also makes it self-documenting and helps administrators easily get a clear picture of their infrastructure configuration. In addition to enforcing configuration end states, Ansible also provides administrators the ability to audit system configurations and detect when they deviate from the desired state.

While configuration management tools like Ansible simplify and improve infrastructure management, they also introduce the dilemma of how this new infrastructure code should be managed. To overcome this challenge, administrators could follow the example of their development counterparts. In accordance with development best practices, administrators should use a version control system, such as Git, to manage their infrastructure code.

Version control allows administrators to implement a life cycle for the different stages of their infrastructure code, such as development, QA, and production. By managing infrastructure

code with a version control tool, administrators can test their infrastructure code changes in noncritical development and QA environments to minimize surprises and disruptions when deployments are implemented in production environments.

## Using Vagrant

When testing code for production deployment, results are only relevant and valid if a test is conducted in a development environment that is identical to the production environment. This is as true for software development as it is for the changes to infrastructure code. Virtualization technology makes it easy and cost-effective to stand up a machine for testing code before production deployment. However, the real challenge is how to construct a development environment on the virtual machine so that it is a replica of the production environment.

The Vagrant software overcomes this challenge by streamlining the creation and configuration of virtual development environments. Vagrant has its own domain-specific language which is used to create a set of instructions for managing virtualization software such as Virtualbox, KVM, and VMware. It can also interact with configuration management software such as Ansible, Puppet, Salt, and Chef. Vagrant simplifies the process of creating and managing consistent virtualized environments needed for development.

Following the Infrastructure-as-Code approach, Vagrant automates the creation of a virtual machine, its hardware configuration, software installation, system configuration, and development source code retrieval by allowing the entire process to be specified within a plain text configuration file. With Vagrant, deploying a development environment can be as simple as checking out a project from version control and executing **vagrant up** on the command line.

An additional benefit of Vagrant's management of virtualized environments as code is that it is very easy not just to create a virtualized development environment, but also to share the identical environment with different team members. Because it insulates the end user from the complexities of setting up and sharing identical virtualized development environments, Vagrant is an ideal tool not only for administrators to test infrastructure code changes but also for developers to test software releases.

### Vagrant components

The Vagrant software is composed of the following main components:

**Vagrant:** The Vagrant software automates the build and configuration of virtual machines. A command line interface is provided to administrators for the management of Vagrant projects. Using the **vagrant** command provided, administrators can instantiate, remove, and manage Vagrant machines. Vagrant is currently not packaged with Red Hat Enterprise Linux but is available for download from the Vagrant website, <https://www.vagrantup.com>.

**Box:** A box is a **tar** file that contains a virtual machine image. A box file serves as the foundation of a Vagrant virtualized environment and is used to create virtual machine instances. For greater flexibility, the image should contain just a base operating system installation. This allows the image to be used as a starting point for creating different virtual machines regardless of the specific requirements of their applications. These application-specific requirements can be fulfilled through automated configuration after the virtual machine is created.

**Provider:** A provider allows Vagrant to interface with the underlying platform that a Vagrant box image is deployed on. Vagrant comes packaged with a provider for Oracle's VirtualBox. Alternative providers are currently available for other virtualization platforms such as VMware, Hyper-V, and KVM.

**Vagrantfile:** Vagrantfile is a plain text file that contains the instructions for creating a Vagrant virtualized environment. The instructions are written using Ruby syntax. The contents of this file can be used to prescribe how the virtual machine is to be built and configured.



## Note

The creator of Vagrant started the HashiCorp company in 2012 to further the development of Vagrant. Administrators can create their own Vagrant box images or leverage existing images publicly available at HashiCorp's Atlas box catalog located at <http://www.vagrantcloud.com>. Preconfigured RHEL 7.1 Vagrant box images for libvirt and Virtual box are available from the Red Hat Container Development Kit. This course will use a preconfigured Red Hat Enterprise Linux box image because the creation of a Vagrant box image is beyond the scope of this course.



## Important

Box files and their contained images are specific to each provider. For example, a box file created for use with the VirtualBox provider is not compatible with the VMware provider. Organizations using multiple providers will need to have separate box files that are specific to each provider.

# Configuring a Vagrant environment

Vagrant environments are meant to be operated in isolation from each other. To create a new Vagrant environment, start by creating a project directory for the new environment. Within this project directory, create a **Vagrantfile** containing the instructions for deploying a new Vagrant machine. The following example shows how developers can initiate Vagrant projects on their workstations.

```
[root@host ~]# mkdir -p /root/vagrant/project
[root@host ~]# cd /root/vagrant/project
[root@host project]# vim Vagrantfile
```

### Creating a basic Vagrantfile

The following lines from a **Vagrantfile** file provide instructions to Vagrant for the creation of a basic virtual machine. The **config.vm.box** method call specifies that the virtual machine be cloned from a box image named **rhel7.1**, which can be obtained from the location specified by the **config.vm.box\_url** method call. The **config.vm.hostname** method call instructs Vagrant to name the virtual machine **sandbox.example.com** when it is created.

```
Vagrant.configure(2) do |config|
 config.vm.box = "rhel7.1"
 config.vm.box_url = "http://content.example.com/ansible2.0/x86_64/dvd/vagrant/rhel-
server-libvirt-7.1-1.x86_64.box"
 config.vm.hostname = "sandbox.example.com"
end
```

### Managing Vagrant machines

With a **Vagrantfile** file created, the Vagrant machine can be instantiated by executing the **vagrant up** command from the root of the project directory.

```
[root@host project]# vagrant up
Bringing machine 'default' up with 'libvirt' provider...
==> default: Box 'rhe17.1' could not be found. Attempting to find and install...
 default: Box Provider: libvirt
 ... Output omitted ...
==> default: Waiting for SSH to become available...
 default:
 default: Vagrant insecure key detected. Vagrant will automatically replace
 default: this with a newly generated keypair for better security.
 default:
 default: Inserting generated public key within guest...
 default: Removing insecure key from the guest if it's present...
 default: Key inserted! Disconnecting and reconnecting using new SSH key...
 ... Output omitted ...
==> default: Setting hostname...
==> default: Configuring and enabling network interfaces...
==> default: Rsyncing folder: /root/vagrant/project/ => /home/vagrant/sync
```

The output of **vagrant up** showcases all the configuration and administration tasks that Vagrant automates and insulates the user from. These tasks include retrieval of the box image, creation of the virtual machine, setting the host name, configuring network interfaces, and copying files from the host to the Vagrant machine. If static IP addressing is not defined in **Vagrantfile**, the virtual machine is dynamically assigned an IP address by the virtualization provider.

When the Vagrant machine has started, it can be accessed using the **vagrant ssh** command. During the deployment of the Vagrant machine, an SSH key is generated on the host and then installed in the `~/.ssh` directory of the **vagrant** user on the Vagrant machine. The **vagrant ssh** command uses this key to authenticate an SSH session to the Vagrant machine as the **vagrant** user.

```
[root@host project]# vagrant ssh
Last login: Wed Oct 21 14:02:44 2015 from 192.168.121.1

[vagrant@sandbox ~]$ id
uid=1000(vagrant) gid=1000(vagrant) groups=1000(vagrant),1001(docker)
context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```



## Note

The Vagrant software expects a **vagrant** user account to be available on the Vagrant machine for SSH access. Therefore, it is standard practice to create the **vagrant** user account and configure its sudo privileges during the creation of base virtual machine images. While SSH key-based authentication is used by Vagrant, it is also standard practice to set the password of the **vagrant** user account to '**vagrant**'.

Having an identical username/password combination goes against good security practice. However, Vagrant environments are meant to be standalone virtual environments hosted on a physical system for development purposes. Having a standard account and password for the management of Vagrant machines also facilitates the use of Vagrant projects by multiple developers.

Vagrant also offers a helpful feature called *synced folders*. By default, Vagrant uses the synchronized folder feature to copy the content of the project directory to a directory on the Vagrant machine (`~/sync/`) that is accessible by the **vagrant** user.

```
[root@host project]# ls -l
total 4
-rw-r--r--. 1 root root 227 Oct 21 13:50 Vagrantfile
```

```
[vagrant@sandbox project]$ ls -l /home/vagrant/sync
total 4
-rw-r--r--. 1 vagrant vagrant 227 Oct 21 13:50 Vagrantfile
```

To allow for the execution of privileged commands on Vagrant machines, box images are built with **sudo** privileges granted to the **vagrant** user. This **sudo** privilege proves especially useful when configuring more advanced Vagrant machine deployments, which will be discussed later.

```
[vagrant@sandbox project]$ sudo -l
Matching Defaults entries for vagrant on this host:
 !visiblepw, always_set_home, env_reset, env_keep="COLORS DISPLAY HOSTNAME
 HISTSIZE INPUTRC KDEDIR LS_COLORS", env_keep+="MAIL PS1 PS2 QTDIR USERNAME
 LANG LC_ADDRESS LC_CTYPE", env_keep+="LC_COLLATE LC_IDENTIFICATION LC_MEASUREMENT
 LC_MESSAGES", env_keep+="LC_MONETARY LC_NAME LC_NUMERIC LC_PAPER LC_TELEPHONE",
 env_keep+="LC_TIME LC_ALL LANGUAGE LINGUAS _XKB_CHARSET XAUTHORITY",
 secure_path=/sbin\:/bin\:/usr/sbin\:/usr/bin

User vagrant may run the following commands on this host:
(ALL) NOPASSWD: ALL
```

When a Vagrant machine is no longer needed, execute the **vagrant halt** command to shut down the Vagrant machine. Another option is to execute the **vagrant destroy** command to stop the running machine, as well as clean up all the resources which were created when the machine was deployed.

```
[vagrant@sandbox project]$ exit
logout
Connection to 192.168.121.85 closed.

[root@host project]# vagrant destroy
==> default: Removing domain...
```

### Vagrant provisioners

The previous example demonstrated the use of Vagrant for quick deployment of a simple virtual machine. Even though this is convenient, the real strength of Vagrant for creating development systems for DevOps environments lies in its provisioning feature.

As mentioned previously, a box image should contain just the base operating system so the image can serve as the foundation for the customization of different virtual machines. Vagrant's provisioning feature automates the software installation and configuration changes needed to overlay the customizations over the base operating system.

Provisioning is performed with the use of one or more *provisioners* offered by Vagrant. Provisioners are enabled in a **Vagrantfile** file with the use of the **config.vm.provision** method call.

Vagrant offers a variety of provisioners to suit the provisioning method preferred by administrators. The most basic provisioner is the *shell provisioner* which uses shell commands to perform provisioning tasks.

The following example demonstrates how Vagrant's shell provisioner can be used to automate the configuration of yum repositories after the Vagrant machine is deployed with a base operating system.

```
Vagrant.configure(2) do |config|
 ...
 config.vm.provision "shell", inline: <<-SHELL
 sudo cp /home/vagrant/sync/etc/yum.repos.d/* /etc/yum.repos.d
 SHELL
 ...
end
```

## References



- Boxes – Vagrant Documentation  
<https://docs.vagrantup.com/v2/boxes.html>
- Providers – Vagrant Documentation  
<https://docs.vagrantup.com/v2/providers/index.html>
- Vagrantfile – Vagrant Documentation  
<https://docs.vagrantup.com/v2/vagrantfile/index.html>
- Vagrant Command-Line Interface – Vagrant Documentation  
<https://docs.vagrantup.com/v2/cli/index.html>
- Vagrant Provisioning – Vagrant Documentation  
<https://docs.vagrantup.com/v2/provisioning/index.html>

# Guided Exercise: Provisioning Vagrant Machines

In this exercise, you will provision a Vagrant machine for use in a DevOps environment.

## Outcomes

You should be able to deploy a Vagrant machine and configure it using the shell provisioner.

## Before you begin

Reset your **tower** server.

## Steps

1. Log in to **workstation** as **student** and run the **ansible-vagrant-practice** lab setup script. The script will make the Vagrant software available on **tower**.

```
[student@workstation ~]$ lab ansible-vagrant-practice setup
```

2. Log in to **tower** as the **root** user. Create a work directory, **/root/vagrant**, for Vagrant work. Also create a subdirectory, **webapp**, inside the Vagrant work directory.

```
[root@tower ~]# mkdir -p vagrant/webapp
```

3. In the **/root/vagrant/webapp** directory, create a Vagrant configuration file, **Vagrantfile** by copying it from **/var/tmp/Vagrantfile**. This file will be used to create a Vagrant machine using a provided box image, name the machine box image **rhel7.1**, and configure the machine with a host name of **dev.lab.example.com**.

```
[root@tower ~]# cd vagrant/webapp
[root@tower webapp]# cp /var/tmp/Vagrantfile .
[root@tower webapp]# cat Vagrantfile
Vagrant.require_version ">= 1.7.0"

Vagrant.configure(2) do |config|
 # Identify which Vagrant box to use
 config.vm.box = "rhel7.1"
 config.vm.box_url = "http://content.example.com/ansible2.0/x86_64/dvd/vagrant/
 rhel-server-libvirt-7.1-1.x86_64.box"

 # Define host settings
 config.vm.hostname = "dev.lab.example.com"

 # Define sync folder(s)

 # Define shell provisioner

 # Define ansible provisioner
end
```

4. Test the deployment of the virtual machine with the new configuration.

```
[root@tower webapp]# vagrant up
Bringing machine 'default' up with 'libvirt' provider...
==> default: Box 'rhel7.1' could not be found. Attempting to find and install...
 default: Box Provider: libvirt
==> default: Creating image (snapshot of base box volume).
==> default: Creating domain with the following settings...
...output omitted...
==> default: Setting hostname...
==> default: Configuring and enabling network interfaces...
==> default: Rsyncing folder: /root/vagrant/webapp/ => /home/vagrant/sync
```

5. Check to see which Yum repositories are available in the Vagrant box.

```
[root@tower webapp]# vagrant ssh
[vagrant@dev ~]$ yum repolist
Loaded plugins: product-id, subscription-manager
repolist: 0
```

6. Exit from and shut down the Vagrant machine. Note that your IP address may be different from the one shown in the example.

```
[vagrant@dev ~]$ exit
logout
Connection to 192.168.121.238 closed.

[root@tower webapp]# vagrant destroy
==> default: Removing domain...
```

7. Because there are no Yum repositories available, configure Vagrant so that the necessary Yum repository configuration files are created when the Vagrant machine is deployed.

- 7.1. Create a directory within the Vagrant work directory to host the necessary Yum repository configuration file and then change to that directory.

```
[root@tower webapp]# mkdir -p /root/vagrant/webapp/etc/yum.repos.d
```

- 7.2. Copy the **/etc/yum.repos.d/rhel\_dvd.repo** Yum repository configuration file to this newly created directory.

```
[root@tower webapp]# cp /etc/yum.repos.d/rhel_dvd.repo /root/vagrant/webapp/etc/yum.repos.d/
```

- 7.3. In the **/root/vagrant/webapp** directory, create a shell script, **provisioner.sh**, which copies Yum repo configuration from the sync directory to the **/etc/yum.repos.d** directory on the Vagrant box. It should have the following contents:

```
#!/bin/bash

Install yum config file
sudo cp /home/vagrant/sync/etc/yum.repos.d/rhel_dvd.repo /etc/yum.repos.d/rhel_dvd.repo
```

The copy must be executed using sudo because it requires **root** privileges.

- 7.4. Modify **Vagrantfile** so that the **provisioner.sh** script is executed by the shell provisioner during provisioning to install the Yum repository configuration file to **/etc/yum.repos.d/rhel\_dvd.repo**. Add the following lines inside the **Vagrant.configure** code block in the **Vagrantfile** file. They should be inserted immediately below the “# Define shell provisioner” comment.

```
Define shell provisioner
config.vm.provision "shell", path: "provisioner.sh"
```

8. Provision the Vagrant machine using the newly modified configuration.

```
[root@tower webapp]# vagrant up
Bringing machine 'default' up with 'libvirt' provider...
==> default: Creating image (snapshot of base box volume).
==> default: Creating domain with the following settings...
...output omitted...
==> default: Rsyncing folder: /root/vagrant/webapp/ => /home/vagrant/sync
==> default: Running provisioner: shell...
 default: Running: /tmp/vagrant-shell20160427-14951-pb7qkg.sh
```

9. Verify that a Yum repository is now available.

```
[root@tower webapp]# vagrant ssh

[vagrant@dev ~]$ yum repolist
Loaded plugins: product-id, subscription-manager
repo id repo name status
rhel_dvd Remote classroom copy of dvd 4,620
repolist: 4,620
```

# Deploying Vagrant in a DevOps Environment

## Objectives

After completing this section, students should be able to:

- Deploy Vagrant in a DevOps environment using Ansible

## Integrating Vagrant with Ansible

In the previous section, deploying a Vagrant machine with a base operating system was shown. Vagrant's provisioning feature and the shell provisioner were demonstrated for the management of configurations on Vagrant machines after their creation from the base operating system image.

In addition to the shell provisioner, Vagrant offers a variety of provisioners to suit the provisioning methodology preferred by an organization. Two types of provisioners are offered by the Vagrant software for administrators who want to manage their Vagrant machines with Ansible.

The *ansible* provisioner performs its work by executing Ansible on the Vagrant host. When using this provisioner, Ansible is installed and executed on the Vagrant host which serves the role of the control node and the Vagrant machines are the managed hosts.

In contrast, the *ansible\_local* provisioner performs its work by executing Ansible on the Vagrant machines. To use this provisioner, Ansible is installed and executed on the Vagrant machine. The Vagrant machine serves as the control node and manages itself as the managed node.

When using Vagrant's ansible provisioner in DevOps environments, it is entirely possible to create a Vagrant machine that is configured identically to production systems. In both cases, the systems are seeded with base operating system installations and then configured through the execution of playbooks. As long as the same playbook used to provision a production system is executed against a Vagrant machine, both systems can be configured identically. The Vagrant machine can then serve as a valid and useful development environment for testing software release or infrastructure code changes before production deployment.

### Using synced folders

Vagrant offers several other features to facilitate the provisioning process. As mentioned previously, one helpful feature is synchronized folders. By default, Vagrant uses the feature to copy contents of the project directory on the host to a directory on the Vagrant machine during its instantiation.

Additional content can be copied from the host to the Vagrant machine by configuring additional synchronized folders in the **Vagrantfile** configuration file. Vagrant offers various mechanisms for keeping these folders synchronized. Vagrant's default synchronization folder type is **VirtualBox**. This folder type offers a bidirectional synchronization of the contents on the host and the Vagrant machine folder, and will continuously synchronize contents as changes are introduced to folder contents on either the host or the Vagrant machine.

**VirtualBox** synchronized folders are only available when using the **VirtualBox** provider. When using another provider, a different synchronization folder type is required. Consult the documentation to understand how each synchronized folder type works, because they can behave differently.

Because this course does not use the **VirtualBox** provider, the **rsync** synchronized folder type will be used instead. Unlike **VirtualBox** synchronized folders, **rsync** synchronized folders are unidirectional and will only copy files from the host to the Vagrant machine and not the other way around. In addition, the **rsync** mechanism does not constantly copy changes to folder contents. It only copies folder contents from the host to the Vagrant machine when **vagrant up** is executed. If folder contents need to be recopied after this initial synchronization, execute the **vagrant rsync** command as shown in the following example.

```
[root@host project]# vagrant rsync
==> default: Rsyncing folder: /root/vagrant/project/ => /home/vagrant/sync
```

### Configuring Vagrant for Ansible provisioning

Like the shell provisioner, the Vagrant Ansible provisioners are employed through the use of the **config.vm.provision** method call in the **Vagrantfile** configuration file. The following example demonstrates how Vagrant's **ansible** provisioner can be used in conjunction with an Ansible playbook, **playbook.yml**, located in the Vagrant project directory, to automate the configuration of a Vagrant machine after it is deployed with a base operating system.

```
Vagrant.configure(2) do |config|
 ...
 config.vm.provision "ansible" do |ansible|
 ansible.playbook = 'playbook.yml'
 end
 ...
end
```

Because the **ansible** provisioner is used, **ansible-playbook** is executed on the Vagrant host. Therefore, the Ansible software must be already be installed on the Vagrant host when this Vagrant Ansible provisioner is used.

In contrast with the **ansible\_local** provisioner, because Ansible is executed on the Vagrant machine, the Ansible software must already be installed on the Vagrant machine prior to the execution of the provisioner. This can be accomplished by installing the Ansible software using another provisioner prior to execution of the **ansible\_local** provisioner.

The following example demonstrates how Vagrant's **ansible\_local** provisioner can be used in conjunction with an Ansible playbook, **playbook.yml**, located in the Vagrant project directory, to automate the configuration of a Vagrant machine after it is deployed with a base operating system. Because the Ansible software is a prerequisite for the successful execution of the **ansible\_local** provisioner, the example uses the **shell** provisioner to first configure a Yum repository and install the Ansible software on the Vagrant machine.

```
Vagrant.configure(2) do |config|
 ...
 config.vm.provision "shell", inline: <<-SHELL
 sudo cp /home/vagrant/sync/etc/yum.repos.d/* /etc/yum.repos.d
 sudo yum install -y ansible
 SHELL
end
```

```

... Configuration omitted ...

config.vm.provision "ansible_local" do |ansible|
 ansible.playbook = 'playbook.yml'
end

... Configuration omitted ...

end

```

## Creating a Vagrant development environment

When a Vagrant machine has been provisioned using a Vagrant Ansible provisioner, it should have all the software needed for its intended purpose. For example, when deploying a Vagrant machine for the development of a web server, one would expect the Ansible playbook used by the Ansible provisioner to install Apache, start the web service, and configure the firewall for HTTP access.

In addition to configuration management, the creation of a development environment can be automated further by incorporating application source code installation into the provisioning playbook. Ansible offers several source control modules to work with version control software, such as Git and Subversion.

The following example shows how the contents of a web application's **DocumentRoot** folder can be installed by calling the **git** module in an Ansible playbook. The task uses the **git** module to clone the contents of the Git repository, **webapp.git**, on **demo.example.com** into the **/var/www/** directory on the Vagrant machine. The **repo** option defines the address of the Git repository. The **dest** option defines the target location for the installation of the source code on the Vagrant machine. The **accept\_hostkey** option is useful when accessing a Git repository using the SSH protocol. It automatically adds the host key for the repository URL.

```

... Configuration omitted ...
- name: get source
 git:
 repo: ssh://student@workstation/home/student/git/webapp.git
 dest: /var/www/html
 accept_hostkey: yes
... Configuration omitted ...

```

### Using forwarded ports

If the application being developed has network services, Vagrant provides a networking configuration feature called *forwarded ports*, which maps network ports on the host system to ports on the Vagrant machine. The following example shows how the **Vagrantfile** file can be modified so that Vagrant forwards traffic directed at port 8000 on the host system to port 80 on the Vagrant machine.

```

Vagrant.configure(2) do |config|
 ... Configuration omitted ...
 config.vm.network :forwarded_port, guest: 80, host: 8000
end

```



## Note

Configuration changes made to **Vagrantfile** will not have any effect on an running Vagrant machine. Changes will take effect when the machine is halted with **vagrant halt** or **vagrant destroy**, and then launched again with **vagrant up**. An alternative is to execute the **vagrant reload** command. This command performs the same actions as **vagrant halt** followed by **vagrant up**. Provisioners defined with the **Vagrantfile** file will not be re-executed when **vagrant reload** is issued.

### Creating a reusable Vagrant development environment

When a Vagrant project is configured to use Ansible in conjunction with a playbook which configures Vagrant machines according to an organization's build standard and also installs application source code, it can be made into a reusable development environment that is easily deployed. Version control systems are widely used by developers to manage application source code. As mentioned previously, one of the advantages of Vagrant's design is that it follows the infrastructure-as-code approach. Because the configuration of a Vagrant machine is maintained as code, it too can also be managed by a version control system. By placing an entire Vagrant project directory in a version control system, such as Git, administrators effectively bundle all the components needed to recreate a preconfigured Vagrant development environment together into a single Git project.

Suppose a new team of developers were tasked to work on the application source code. These developers merely need to install the Vagrant software on their workstations and then run **git clone** followed by **vagrant up**. The **git clone** command retrieves all the Vagrant project components (Vagrant configuration, configuration files, playbooks, and so on). The **vagrant up** recreates the Vagrant development environment using Ansible and the retrieved components.

Because the recreation of the development environment is dictated by coded instructions, each developer ends up with a development environment that is not only identical to that on each of their teammates' workstations, but also identical to that on the production server. This provides the developers assurance that application source code changes validated on the Vagrant development environments on their workstations will behave identically when deployed to production. Perhaps the most elegant part of this design is that no work was required on the part of the operations staff to get these developers up and running.

The intelligence of this design becomes even more evident when changes are required by the operations team. As operations staff make changes to the production environment, such as deploying new software versions to address bugs or security vulnerabilities, the same changes need to be made to the associated development environments to keep them identical. This is accomplished quickly and easily by implementing the production playbook changes to the playbook used for the provisioning of the Vagrant development machines. The Ansible provisioner applies the new configuration when new Vagrant machines are instantiated. This has little to no impact on the developers, who are free to continue with development tasks.

A development environment implemented in this manner with Vagrant allows both development and operations teams to operate in cooperation rather than in conflict with each other. Developers are empowered to easily deploy development environments that are replicas of the production environment by themselves without burdening the operations team. Because the operations team have control over development environments' resemblance to the production environment, code deployments to production should be uneventful.



## References

- Ansible and Vagrant – Vagrant Documentation**  
[https://www.vagrantup.com/docs/provisioning/ansible\\_intro.html](https://www.vagrantup.com/docs/provisioning/ansible_intro.html)
- Ansible Provisioner – Vagrant Documentation**  
<https://www.vagrantup.com/docs/provisioning/ansible.html>
- Ansible Local Provisioner – Vagrant Documentation**  
[https://www.vagrantup.com/docs/provisioning/ansible\\_local.html](https://www.vagrantup.com/docs/provisioning/ansible_local.html)
- Vagrant Rsync Synced Folder – Vagrant Documentation**  
<https://docs.vagrantup.com/v2/synced-folders/rsync.html>
- Vagrant Forwarded Ports – Vagrant Documentation**  
[https://docs.vagrantup.com/v2/networking/forwarded\\_ports.html](https://docs.vagrantup.com/v2/networking/forwarded_ports.html)

# Guided Exercise: Deploying Vagrant in a DevOps Environment

In this exercise, you will create a development environment using Vagrant and Ansible.

## Outcomes

You should be able to use Ansible to deploy a Vagrant machine as a development environment for a website.

### Before you begin

**tower** should have a working Vagrant machine configured with a YUM repository as covered in a previous exercise.

### Steps

1. Log in to **workstation** and run **lab ansible-devops-practice setup**. The lab setup script will install and configure the Git on the repository server and on **tower**. It also downloads the playbook needed for this exercise.

```
[student@workstation ~]$ lab ansible-devops-practice setup
```

2. Log in to **tower** as the **root** user. Change directory to **/root/vagrant/webapp** and verify the status of the Vagrant environment.
  - 2.1. The Vagrant machine from the previous exercise should not be running.

```
[root@tower ~]# cd vagrant/webapp
[root@tower webapp]# vagrant status
Current machine states:

default not created (libvirt)

The Libvirt domain is not created. Run `vagrant up` to create it.
```

- 2.2. If the Vagrant machine from the previous exercise is running, terminate the instance.

```
[root@tower webapp]# vagrant status
Current machine states:

default running (libvirt)

The Libvirt domain is running. To stop this machine, you can run
`vagrant halt`. To destroy the machine, you can run `vagrant destroy`.

[root@tower webapp]# vagrant destroy
==> default: Removing domain...
==> default: Running cleanup tasks for 'shell' provisioner...
```

3. Create a playbook, **intranet-dev.yml** by copying it from **/var/tmp/intranet-dev.yml**. This playbook will be used with the Vagrant **ansible** provisioner to provision the Vagrant machine as a website development environment.

```
[root@tower webapp]# cp /var/tmp/intranet-dev.yml .
```

4. Review the downloaded playbook to determine the tasks that will be performed. The playbook will ensure that the latest versions of the *httpd* and *firewalld* packages are installed and that **firewalld** is configured to allow incoming connections for the http service. It also installs SSH private key and Git configuration files which will then be used to obtain the source code for the intranet website.

```
- name: intranet services
hosts: all
become: yes
become_user: root
tasks:
- name: latest httpd version installed
 yum:
 name: httpd
 state: latest
- name: latest firewalld version installed
 yum:
 name: firewalld
 state: latest
- name: httpd enabled and running
 service:
 name: httpd
 enabled: true
 state: started
- name: firewalld enabled and running
 service:
 name: firewalld
 enabled: true
 state: started
- name: firewalld permits http service
 firewalld:
 service: http
 permanent: true
 state: enabled
 notify:
 - restart firewalld
- name: create .ssh directory
 file:
 path: /root/.ssh
 state: directory
 mode: 0700
- name: install private key file
 copy:
 src: /root/.ssh/id_rsa
 dest: /root/.ssh/id_rsa
 mode: 0600
- name: install git configuration
 copy:
 src: /root/.gitconfig
 dest: /root/.gitconfig
 mode: 0644
- name: get source
 git:
 repo: ssh://student@workstation/home/student/git/webapp.git
 dest: /var/www/html
 accept_hostkey: yes
handlers:
- name: restart firewalld
```

```
service:
 name: firewalld
 state: restarted
```

5. Modify the Vagrant configuration file to use the **ansible** provisioner to configure the Vagrant machine to host a website development environment.
  - 5.1. Modify the Vagrant configuration file so that the **ansible** provisioner is used to provision the Vagrant machine using the downloaded playbook, **intranet-dev.yml**.

```
Define ansible provisioner
config.vm.provision "ansible" do |ansible|
 ansible.playbook = "intranet-dev.yml"
end
end
```

- 5.2. Modify the Vagrant configuration file host settings so port **80** of the Vagrant machine can be accessed on port **8000** on **localhost**. Add the following line within the **Vagrant.configure** code block.

```
Define host settings
config.vm.hostname = "dev.lab.example.com"
config.vm.network "forwarded_port", guest: 80, host: 8000
```

6. Install Ansible on the Vagrant host, **tower**, so that it is available for use by Vagrant's **ansible** provisioner.

```
[root@tower webapp]# yum install -y ansible
```

7. Bring the Vagrant machine up. It will use the newly introduced changes.

```
[root@tower webapp]# vagrant up
Bringing machine 'default' up with 'libvirt' provider...
==> default: Creating image (snapshot of base box volume).
==> default: Creating domain with the following settings...
...output omitted...
==> default: Running provisioner: ansible...

PLAY [intranet services for development] ****
TASK [Gathering Facts] ****
ok: [default]

TASK [latest httpd version installed] ****
changed: [default]

TASK [latest firewalld version installed] ****
changed: [default]

TASK [httpd enabled and started] ****
changed: [default]

TASK [firewalld enabled and started] ****
changed: [default]

TASK [firewalld permits http service] ****
```

```
changed: [default]

TASK [create .ssh directory] ****
changed: [default]

TASK [install private key file] ****
changed: [default]

TASK [install git configuration] ****
changed: [default]

TASK [get source] ****
changed: [default]

RUNNING HANDLER [restart firewalld] ****
changed: [default]

PLAY RECAP ****
default : ok=11 changed=10 unreachable=0 failed=0
```

8. Verify that the web application is working properly on the Vagrant machine using port 8000 on **localhost**.

```
[root@tower webapp]# curl http://localhost:8000
Welcome to Web App 1.0
```

# Lab: Implementing Ansible in a DevOps Environment

In this lab, you will modify and publish web content using Vagrant and Ansible.

## Outcomes

You should be able modify web content on a development website running on a Vagrant machine and deploy the changes to a production server using Ansible.

### Before you begin

The development team you just joined is hosting their intranet development web server on a Vagrant machine. Vagrant uses the **ansible** provisioner in conjunction with the **intranet-dev.yml** file to provision this Vagrant machine.

In addition, to streamline the process of pushing web content to the intranet production web server, **servera**, the team is making use of an **ansible.cfg** configuration file, **inventory** inventory file, and an **intranet-prod.yml** playbook. They are storing everything in a Git repository so that a new team member can just check out the project and have everything they need to work on the development web server and then push code to the production web server.

You have been assigned the task of updating the **/var/www/html/index.html** so that it contains the message "**Welcome to Web App 2.0**".

Before you begin, reset **tower**. Then, log in to **workstation** and run **lab ansible-vagrant-lab setup** to install the Vagrant software, create the Git repository, as well as install and configure the Git software.

```
[student@workstation ~]$ lab ansible-vagrant-lab setup
```

### Steps

1. Log in to **tower** as the **root** user and install the Ansible software so it will be available for use by the Vagrant ansible provisioner.
2. Create a working directory for the Vagrant web application environment.
3. Clone the Vagrant intranet development environment from the Git repository on **workstation** using the following command.

```
[root@tower webapp]# git clone student@workstation:/var/git/vagrantwebapp.git .
```

Start the Vagrant machine that hosts the intranet development web server.

- 3.1. Clone the Vagrant environment from the **vagrantwebapp.git** Git repo.
- 3.2. Start the Vagrant machine.
4. Connect to and verify the contents on the development intranet website from the Vagrant host, **tower**.

```
[root@tower webapp]# curl http://localhost:8000
```

Welcome to Web App 1.0

5. Update the development intranet website by modifying the **/var/www/html/index.html** file on the Vagrant machine. Commit the change and push the changes to the Git repository using the following commands executed from **/var/www/html** directory on the Vagrant machine.

```
[root@dev html]# git commit -am 'New web app version'
[root@dev html]# git push origin master
```

Exit the Vagrant machine after the changes are pushed.

6. Connect to and verify the changes made on the development intranet website from the Vagrant host, **tower**.

```
[root@tower webapp]# curl http://localhost:8000
Welcome to Web App 2.0
```

7. Deploy the changes to the production intranet website by executing the playbook, **intranet-prod.yml** using the **ansible.cfg** configuration and **inventory** file.
8. Connect to and verify that the changes made on the development intranet website have been propagated to the production intranet website running on **servera**.
9. Run **lab ansible-vagrant-lab grade** on **workstation** to grade your work.

```
[student@workstation ~]$ lab ansible-vagrant-lab grade
```

## Solution

In this lab, you will modify and publish web content using Vagrant and Ansible.

### Outcomes

You should be able to modify web content on a development website running on a Vagrant machine and deploy the changes to a production server using Ansible.

#### Before you begin

The development team you just joined is hosting their intranet development web server on a Vagrant machine. Vagrant uses the **ansible** provisioner in conjunction with the **intranet-dev.yml** file to provision this Vagrant machine.

In addition, to streamline the process of pushing web content to the intranet production web server, **servera**, the team is making use of an **ansible.cfg** configuration file, **inventory** inventory file, and an **intranet-prod.yml** playbook. They are storing everything in a Git repository so that a new team member can just check out the project and have everything they need to work on the development web server and then push code to the production web server.

You have been assigned the task of updating the `/var/www/html/index.html` so that it contains the message "**Welcome to Web App 2.0**".

Before you begin, reset **tower**. Then, log in to **workstation** and run **lab ansible-vagrant-lab setup** to install the Vagrant software, create the Git repository, as well as install and configure the Git software.

```
[student@workstation ~]$ lab ansible-vagrant-lab setup
```

#### Steps

1. Log in to **tower** as the **root** user and install the Ansible software so it will be available for use by the Vagrant ansible provisioner.

```
[root@tower ~]# yum install -y ansible
Loaded plugins: langpacks, search-disabled-repos
Resolving Dependencies
--> Running transaction check
...output omitted
Dependency Installed:
 libgnome-keyring.x86_64 0:3.8.0-3.el7 perl-Error.noarch 1:0.17020-2.el7
 perl-Git.noarch 0:1.8.3.1-5.el7 python-crypto.x86_64 0:2.6-4.el7
 perl-TermReadKey.x86_64 0:2.30-20.el7 python-ecdsa.noarch 0:0.11-3.el7ost
 python-httplib2.noarch 0:0.7.7-3.el7 python-keyczar.noarch 0:0.71c-2.el7
 python-paramiko.noarch 0:1.15.1-1.el7 sshpass.x86_64 0:1.05-1.el7.rf
 python-pyasn1.noarch 0:0.1.6-2.el7

Complete!
```

2. Create a working directory for the Vagrant web application environment.

```
[root@tower ~]# mkdir -p vagrant/webapp
[root@tower ~]# cd vagrant/webapp
```

3. Clone the Vagrant intranet development environment from the Git repository on **workstation** using the following command.

```
[root@tower webapp]# git clone student@workstation:/var/git/vagrantwebapp.git .
```

Start the Vagrant machine that hosts the intranet development web server.

3.1. Clone the Vagrant environment from the **vagrantwebapp.git** Git repo.

```
[root@tower webapp]# git clone student@workstation:/var/git/vagrantwebapp.git .
Cloning into 'vagrantwebapp'...
The authenticity of host 'workstation (172.25.250.254)' can't be established.
ECDSA key fingerprint is 84:fc:5e:82:a8:4f:bb:f3:c0:06:61:77:ad:a5:e5:b9.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'workstation,172.25.250.254' (ECDSA) to the list of
known hosts.
remote: Counting objects: 16, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 16 (delta 1), reused 16 (delta 1)
Receiving objects: 100% (16/16), done.
Resolving deltas: 100% (1/1), done.
```

3.2. Start the Vagrant machine.

```
[root@tower webapp]# vagrant up
Bringing machine 'default' up with 'libvirt' provider...
==> default: Box 'rhel7.1' could not be found. Attempting to find and install...
 default: Box Provider: libvirt
 ...output omitted
RUNNING HANDLER [restart firewalld] ****
changed: [default]

PLAY RECAP ****
default : ok=11 changed=10 unreachable=0 failed=0
```

4. Connect to and verify the contents on the development intranet website from the Vagrant host, **tower**.

```
[root@tower webapp]# curl http://localhost:8000
Welcome to Web App 1.0
```

5. Update the development intranet website by modifying the **/var/www/html/index.html** file on the Vagrant machine. Commit the change and push the changes to the Git repository using the following commands executed from **/var/www/html** directory on the Vagrant machine.

```
[root@dev html]# git commit -am 'New web app version'
[root@dev html]# git push origin master
```

Exit the Vagrant machine after the changes are pushed.

5.1. Log in to the Vagrant machine.

```
[root@tower webapp]# vagrant ssh
Last login: Fri Apr 29 10:30:04 2016 from 192.168.121.1
```

- 5.2. Become the **root** user.

```
[vagrant@dev ~]$ sudo -i
```

- 5.3. Change to the development website's document root directory.

```
[root@dev ~]# cd /var/www/html
```

- 5.4. Modify the **index.html** file so it contains the message "**Welcome to Web App 2.0**".

```
Welcome to Web App 2.0
```

- 5.5. Commit and push the change to the Git repository.

```
[root@dev html]# git commit -am 'New web app version'
 1 file changed, 1 insertion(+), 1 deletion(-)
[root@dev html]# git push origin master
Counting objects: 5, done.
Writing objects: 100% (3/3), 262 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To ssh://student@workstation/var/git/webapp.git
 1d0ad93..6f75595 master -> master
```

- 5.6. Exit the Vagrant machine.

```
[root@dev html]# exit
logout
[vagrant@dev ~]$ exit
logout
Connection to 192.168.121.56 closed.
```

6. Connect to and verify the changes made on the development intranet website from the Vagrant host, **tower**.

```
[root@tower webapp]# curl http://localhost:8000
Welcome to Web App 2.0
```

7. Deploy the changes to the production intranet website by executing the playbook, **intranet-prod.yml** using the **ansible.cfg** configuration and **inventory** file.

```
[root@tower webapp]# cat ansible.cfg
[defaults]
inventory = inventory
host_key_checking = False
[root@tower webapp]# cat inventory
servera.lab.example.com
[root@tower webapp]# ansible-playbook intranet-prod.yml

PLAY [intranet services for production] ****
TASK [Gathering Facts] ****
ok: [servera.lab.example.com]
```

```
TASK [latest httpd version installed] ****
ok: [servera.lab.example.com]

TASK [latest firewalld version installed] ****
ok: [servera.lab.example.com]

TASK [httpd enabled and started] ****
ok: [servera.lab.example.com]

TASK [firewalld enabled and started] ****
ok: [servera.lab.example.com]

TASK [firewalld permits http service] ****
ok: [servera.lab.example.com]

TASK [get source] ****
changed: [servera.lab.example.com]

PLAY RECAP ****
servera.lab.example.com : ok=7 changed=1 unreachable=0 failed=0
```

8. Connect to and verify that the changes made on the development intranet website have been propagated to the production intranet website running on **servera**.

```
[root@tower webapp]# curl http://servera
Welcome to Web App 2.0
```

9. Run **lab ansible-vagrant-lab grade** on **workstation** to grade your work.

```
[student@workstation ~]$ lab ansible-vagrant-lab grade
```

## Summary

In this chapter, you learned:

- The Vagrant software requires a box, a provider, and a **Vagrantfile** to create a Vagrant machine.
- **Vagrantfile** is used to configure a Vagrant machine.
- Vagrant provisioners automate software installation and system configuration after a Vagrant machine has been built from a box image.
- Vagrant's **ansible** provisioner automates the provisioning of a Vagrant machine through execution of Ansible on the Vagrant host.
- Vagrant's **ansible\_local** provisioner automates the provisioning of a Vagrant machine through execution of Ansible on the Vagrant machine.



## CHAPTER 13

# COMPREHENSIVE REVIEW: AUTOMATION WITH ANSIBLE

Overview	
<b>Goal</b>	Demonstrate skills learned in this course by installing, optimizing, and configuring Ansible for the management of managed hosts.
<b>Sections</b>	<ul style="list-style-type: none"><li>• Comprehensive Review</li></ul>
<b>Lab</b>	<ul style="list-style-type: none"><li>• Lab: Deploying Ansible</li><li>• Lab: Creating Playbooks</li><li>• Lab: Creating Roles and using Dynamic Inventories</li><li>• Lab: Optimizing Ansible</li><li>• Lab: Deploying Ansible Tower and Executing Jobs</li></ul>

# Comprehensive Review

## Objective

After completing this section, students should be able to demonstrate proficiency with knowledge and skills learned in *Automation with Ansible*.

## Reviewing *Automation with Ansible*

Before beginning the comprehensive review for this course, students should be comfortable with the topics covered in each chapter.

Students can refer to earlier sections in the textbook for extra study.

### *Chapter 1, Introducing Ansible*

Describe the terminology and architecture of Ansible.

- Describe Ansible concepts, reference architecture, and use cases.
- Install Ansible.

### *Chapter 2, Deploying Ansible*

Configure Ansible and run ad hoc commands.

- Describe Ansible inventory concepts and build a static inventory.
- Manage Ansible configuration files.
- Run Ansible ad hoc commands.
- Manage dynamic inventory.

### *Chapter 3, Implementing Playbooks*

Write Ansible plays and execute a playbook.

- Write a basic Ansible Playbook and run it using the **ansible-playbook** command.
- Write and run a more sophisticated Ansible Playbook using multiple plays and privilege escalation.

### *Chapter 4, Managing Variables and Inclusions*

To describe variable scope and precedence, manage variables and facts in a play, and manage inclusions.

- Manage variables in Ansible projects
- Manage Facts in Playbooks
- Include variables and tasks from external files into a playbook

### *Chapter 5, Implementing Task Control*

Manage task control, handlers, and tags in Ansible playbooks

- Construct conditionals and loops in a playbook
- Implement handlers in a playbook
- Implement tags in a playbook

- Resolve errors in a playbook

### ***Chapter 6, Implementing Jinja2 Templates***

Implement a Jinja2 template

- Describe Jinja2 templates
- Implement Jinja2 templates

### ***Chapter 7, Implementing Roles***

Create and manage roles

- Describe the structure and behavior of a role
- Create a role
- Deploy roles with Ansible Galaxy

### ***Chapter 8, Optimizing Ansible***

Tune how Ansible executes plays and tasks using host patterns, delegation, and parallelism

- Specify managed hosts for plays and ad hoc commands using host patterns
- Configure delegation in a playbook
- Configure parallelism in Ansible

### ***Chapter 9, Implementing Ansible Vault***

Manage encryption with Ansible Vault.

- Create, edit, rekey, encrypt, and decrypt files.
- Run a playbook with Ansible Vault.



### **Important**

The material on Ansible Vault is not revisited in this chapter. Students wishing to review it, especially in preparation for the EX407 certification exam, should look at the exercises in *Chapter 9, Implementing Ansible Vault*.

### ***Chapter 10, Troubleshooting Ansible***

Troubleshoot playbooks and managed hosts.

- Troubleshoot playbooks
- Troubleshoot managed hosts

### ***Chapter 11, Implementing Ansible Tower***

Explain what Ansible Tower is and demonstrate a basic ability to navigate and use its web user interface.

- Describe the architecture, use cases, and installation requirements of Ansible Tower.
- Install a new Ansible Tower on a single node using the setup.sh script.
- Navigate and describe the Ansible Tower web user interface, and successfully launch a job using the demo job template, project, credential, and inventory.

***Chapter 12, Implementing Ansible in a DevOps Environment***

Implement Ansible in a DevOps environment using Vagrant.

- Describe Ansible in a DevOps environment and provision Vagrant machines.
- Deploy Vagrant in a DevOps environment.

# Lab: Deploying Ansible

In this review, you will install Ansible on **workstation** and use it as a control node and configure it for connections to the managed hosts **servera** and **serverb**. Use ad hoc commands to perform actions on the managed hosts.

## Outcomes

You should be able to:

- Install Ansible.
- Use ad hoc commands to perform actions on managed hosts.

## Before you begin

Save any work that you want to keep on your machines. When you have saved any data you want to keep, reset all of the virtual machines.

If you did not reset all of your virtual machines at the end of the last lab, do so now. This may take a few minutes.

Log in as the **student** user on **workstation** and run **lab ansible-deploy-cr setup**. This script ensures that the managed hosts, **servera** and **serverb**, are reachable on the network. The script creates a directory structure for the lab in the student's home directory.

```
[student@workstation ~]$ lab ansible-deploy-cr setup
```

## Instructions

Install and configure Ansible on **workstation**, and ensure that you meet the following criteria. Demonstrate that you can construct the ad hoc commands specified in the list of criteria in order to modify the managed hosts and verify that the modifications work as expected.

- Install Ansible on **workstation** so that it can serve as the control node.
- On the control node, create an inventory file, **/home/student/ansible-deploy-cr/inventory/hosts**, containing a group called **dev**. This group should consist of the managed hosts **servera.lab.example.com** and **serverb.lab.example.com**.
- Create the Ansible configuration file in **/home/student/ansible-deploy-cr/ansible.cfg**. The configuration file should point to the inventory file created in **/home/student/ansible-deploy-cr/inventory**.
- Execute an ad hoc command using privilege escalation to modify the contents of the **/etc/motd** file on **servera** and **serverb** so that it contains the string **Managed by Ansible\n**. Use **devops** as the remote user.
- Execute an ad hoc command to verify that the contents of the **/etc/motd** file on **servera** and **serverb** are identical.

## Evaluation

From **workstation**, run the **lab ansible-deploy-cr** script with the **grade** argument to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab ansible-deploy-cr grade
```

# Solution

In this review, you will install Ansible on **workstation** and use it as a control node and configure it for connections to the managed hosts **servera** and **serverb**. Use ad hoc commands to perform actions on the managed hosts.

## Outcomes

You should be able to:

- Install Ansible.
- Use ad hoc commands to perform actions on managed hosts.

## Before you begin

Save any work that you want to keep on your machines. When you have saved any data you want to keep, reset all of the virtual machines.

If you did not reset all of your virtual machines at the end of the last lab, do so now. This may take a few minutes.

Log in as the **student** user on **workstation** and run **lab ansible-deploy-cr setup**. This script ensures that the managed hosts, **servera** and **serverb**, are reachable on the network. The script creates a directory structure for the lab in the student's home directory.

```
[student@workstation ~]$ lab ansible-deploy-cr setup
```

## Instructions

Install and configure Ansible on **workstation**, and ensure that you meet the following criteria. Demonstrate that you can construct the ad hoc commands specified in the list of criteria in order to modify the managed hosts and verify that the modifications work as expected.

- Install Ansible on **workstation** so that it can serve as the control node.
- On the control node, create an inventory file, **/home/student/ansible-deploy-cr/inventory/hosts**, containing a group called **dev**. This group should consist of the managed hosts **servera.lab.example.com** and **serverb.lab.example.com**.
- Create the Ansible configuration file in **/home/student/ansible-deploy-cr/ansible.cfg**. The configuration file should point to the inventory file created in **/home/student/ansible-deploy-cr/inventory**.
- Execute an ad hoc command using privilege escalation to modify the contents of the **/etc/motd** file on **servera** and **serverb** so that it contains the string **Managed by Ansible\n**. Use **devops** as the remote user.
- Execute an ad hoc command to verify that the contents of the **/etc/motd** file on **servera** and **serverb** are identical.

## Steps

1. Install Ansible on **workstation** so that it can serve the control node.

```
[student@workstation ~]$ sudo yum -y install ansible
We trust you have received the usual lecture from the local System
Administrator. It usually boils down to these three things:
```

```
#1) Respect the privacy of others.
#2) Think before you type.
#3) With great power comes great responsibility.
```

```
[sudo] password for student: student
```

2. On the control node, create an inventory file, **/home/student/ansible-deploy-cr/inventory/hosts**, containing a group called **dev**. This group should consist of the managed hosts **servera.lab.example.com** and **serverb.lab.example.com**.

- 2.1. Use the **vim** text editor to create and edit the inventory file **/home/student/ansible-deploy-cr/inventory/hosts**.

```
[student@workstation ~]$ cd /home/student/ansible-deploy-cr/inventory/
[student@workstation inventory]$ vim hosts
```

- 2.2. Add the following entries to the file to create the **dev** host group and members **servera.lab.example.com** and **serverb.lab.example.com**. Save the changes and exit the text editor.

```
[dev]
servera.lab.example.com
serverb.lab.example.com
```

3. Create the Ansible configuration file in **/home/student/ansible-deploy-cr/ansible.cfg**. The configuration file should point to the inventory file created in **/home/student/ansible-deploy-cr/inventory**.

- 3.1. Use the **vim** text editor to create and edit the ansible configuration file **/home/student/ansible-deploy-cr/ansible.cfg**.

```
[student@workstation inventory]$ cd ..
[student@workstation ansible-deploy-cr]$ vim ansible.cfg
```

- 3.2. Add the following entries to the file to point to the inventory directory **/home/student/ansible-deploy-cr/inventory**. Save the changes and exit the text editor.

```
[defaults]
inventory=/home/student/ansible-deploy-cr/inventory
```

4. Execute an ad hoc command using privilege escalation to modify the contents of the **/etc/motd** file on **servera** and **serverb** so that it contains the string **Managed by Ansible\n**. Use **devops** as the remote user.

- 4.1. From the directory **/home/student/ansible-deploy-cr**, execute an ad hoc command using privilege escalation to modify the contents of the **/etc/motd** file on **servera** and **serverb**, so that it contains the string **Managed by Ansible\n**. Use **devops** as the remote user.

```
[student@workstation ansible-deploy-cr]$ ansible dev -m copy -a
'content="Managed by Ansible\n" dest=/etc/motd' -b -u devops
```

```

serverb.lab.example.com | SUCCESS => {
 "changed": true,
 "checksum": "4458b979ede3c332f8f2128385df4ba305e58c27",
 "dest": "/etc/motd",
 "gid": 0,
 "group": "root",
 "md5sum": "65a4290ee5559756ad04e558b0e0c4e3",
 "mode": "0644",
 "owner": "root",
 "secontext": "system_u:object_r:etc_t:s0",
 "size": 19,
 "src": "/home/devops/.ansible/tmp/ansible-tmp-1463700139.62-268323083587449/
source",
 "state": "file",
 "uid": 0
}
servera.lab.example.com | SUCCESS => {
 "changed": true,
 "checksum": "4458b979ede3c332f8f2128385df4ba305e58c27",
 "dest": "/etc/motd",
 "gid": 0,
 "group": "root",
 "md5sum": "65a4290ee5559756ad04e558b0e0c4e3",
 "mode": "0644",
 "owner": "root",
 "secontext": "system_u:object_r:etc_t:s0",
 "size": 19,
 "src": "/home/devops/.ansible/tmp/ansible-tmp-1463700139.63-258952343613886/
source",
 "state": "file",
 "uid": 0
}

```

- Execute an ad hoc command to verify that the contents of the **/etc/motd** file on **servera** and **serverb** are identical.

```

[student@workstation ansible-deploy-cr]$ ansible dev -m command -a "cat /etc/motd"
servera.lab.example.com | SUCCESS | rc=0 >>
Managed by Ansible

serverb.lab.example.com | SUCCESS | rc=0 >>
Managed by Ansible

```

#### Evaluation

From **workstation**, run the **lab ansible-deploy-cr** script with the **grade** argument to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab ansible-deploy-cr grade
```

# Lab: Creating Playbooks

In this review, you will create three playbooks in the Ansible project directory, **/home/student/ansible-playbooks-cr**. One playbook will ensure that *lftp* is installed on systems that should be FTP clients, one playbook will ensure that *vsftpd* is installed and configured on systems that should be FTP servers, and one playbook (**site.yml**) will run both of the other playbooks.

## Outcomes

You should be able to:

- Create and execute playbooks to perform tasks on managed hosts.
- Utilize Jinja2 templates, variables, and handlers in playbooks.

## Before you begin

Set up your computers for this exercise by logging into **workstation** as **student**, and run the following command:

```
[student@workstation ~]$ lab ansible-playbooks-cr setup
```

## Instructions

Create a static inventory in **ansible-playbooks-cr/inventory/hosts** with **serverc.lab.example.com** in the group **ftpclients**, and **serverb.lab.example.com** and **serverd.lab.example.com** in the group **ftpservers**. Create an **ansible-playbooks-cr/ansible.cfg** file which configures your Ansible project to use this inventory. (You may find it useful to look at the system's **/etc/ansible/ansible.cfg** file for help with syntax.)

Configure your Ansible project to connect to hosts in the inventory using the remote user, **devops**, and the **sudo** method for privilege escalation. You have SSH keys to log in as **devops** already configured. The **devops** user does not need a password for privilege escalation with **sudo**.

Create a playbook named **ftpclient.yml** in the **ansible-playbooks-cr** directory that contains a play targeting hosts in the inventory group **ftpclients**. It should make sure the *lftp* package is installed.

Create a second playbook named **ansible-vsftpd.yml** in the **ansible-playbooks-cr** directory that contains a play targeting hosts in the inventory group **ftpservers**. It should be written as follows:

- You have a configuration file for **vsftpd** generated from a Jinja2 template. Create a directory for templates, **ansible-playbooks-cr/templates**, and copy the provided **vsftpd.conf.j2** file into it. Also create the directory **ansible-playbooks-cr/vars**. Copy into that directory the provided **defaults-template.yml** file, which contains default variable settings used to complete that template when it is deployed.
- Create a variable file, **ansible-playbooks-cr/vars/vars.yml**, that sets three variables:

Variable	Value
vsftpd_packages	vsftpd

Variable	Value
vsftpd_service	vsftpd
vsftpd_config_file	/etc/vsftpd/vsftpd.conf

- In your **ansible-vsftpd.yml** playbook, make sure that you use **vars\_files** to include the files of variables in the **ansible-playbooks-cr/vars** directory in your play.
- In the play in **ansible-vsftpd.yml**, create tasks which:
  1. Ensure that the packages listed by the variable **{{ vsftpd\_packages }}** are installed.
  2. Ensure that the services listed by the variable **{{ vsftpd\_service }}** are started and enabled to start at boot time.
  3. Use the **template** module to deploy the **templates/vsftpd.conf.j2** template to the location defined by the **{{ vsftpd\_config\_file }}** variable. The file should be owned by user **root**, group **root**, have octal file permissions **0600**, and an SELinux type of **etc\_t**. Notify a handler that restarts **vsftpd** if this task cause a change.
  4. Ensure that the *firewalld* package is installed and that the service is started and enabled. Ensure that **firewalld** has been configured to immediately and permanently allow connections to the ftp service.
- In your **ansible-vsftpd.yml** playbook, create a handler to restart the services listed by the variable **{{ vsftpd\_service }}** when notified.

Create a third playbook, **site.yml**, in the **ansible-playbooks-cr** directory. This playbook should include the plays from the other two playbooks by containing exactly two lines:

```
- include: ansible-vsftpd.yml
- include: ftpclients.yml
```

You are encouraged to follow recommended playbook practices by naming all your plays and tasks. The playbooks should be written using appropriate modules, and should be able to be rerun safely. The playbooks should not make unnecessary changes to the systems.

Remember to use the **ansible-doc** command to help you find modules and information on how to use them.

When done, you should use **ansible-playbook site.yml** to check your work before running the grading script. You may also run the individual playbooks separately to make sure that they function.



## Important

If you are having trouble with your **site.yml** playbook, make sure that both **ansible-vsftpd.yml** and **ftpclients.yml** have indentation consistent with each other.

### Evaluation

As the **student** user on **workstation**, run the **lab ansible-playbooks-cr grade** command to confirm success of this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab ansible-playbooks-cr grade
```

### Cleanup

Run the **lab ansible-playbooks-cr cleanup** command to clean up the lab tasks on **serverb**, **serverc**, and **serverd**.

```
[student@workstation ~]$ lab ansible-playbooks-cr cleanup
```

# Solution

In this review, you will create three playbooks in the Ansible project directory, `/home/student/ansible-playbooks-cr`. One playbook will ensure that `lftp` is installed on systems that should be FTP clients, one playbook will ensure that `vsftpd` is installed and configured on systems that should be FTP servers, and one playbook (`site.yml`) will run both of the other playbooks.

## Outcomes

You should be able to:

- Create and execute playbooks to perform tasks on managed hosts.
- Utilize Jinja2 templates, variables, and handlers in playbooks.

## Before you begin

Set up your computers for this exercise by logging into `workstation` as `student`, and run the following command:

```
[student@workstation ~]$ lab ansible-playbooks-cr setup
```

### Instructions

Create a static inventory in `ansible-playbooks-cr/inventory/hosts` with `serverc.lab.example.com` in the group `ftpclients`, and `serverb.lab.example.com` and `serverd.lab.example.com` in the group `ftpservers`. Create an `ansible-playbooks-cr/ansible.cfg` file which configures your Ansible project to use this inventory. (You may find it useful to look at the system's `/etc/ansible/ansible.cfg` file for help with syntax.)

Configure your Ansible project to connect to hosts in the inventory using the remote user, `devops`, and the `sudo` method for privilege escalation. You have SSH keys to log in as `devops` already configured. The `devops` user does not need a password for privilege escalation with `sudo`.

Create a playbook named `ftpclient.yml` in the `ansible-playbooks-cr` directory that contains a play targeting hosts in the inventory group `ftpclients`. It should make sure the `lftp` package is installed.

Create a second playbook named `ansible-vsftpd.yml` in the `ansible-playbooks-cr` directory that contains a play targeting hosts in the inventory group `ftpservers`. It should be written as follows:

- You have a configuration file for `vsftpd` generated from a Jinja2 template. Create a directory for templates, `ansible-playbooks-cr/templates`, and copy the provided `vsftpd.conf.j2` file into it. Also create the directory `ansible-playbooks-cr/vars`. Copy into that directory the provided `defaults-template.yml` file, which contains default variable settings used to complete that template when it is deployed.
- Create a variable file, `ansible-playbooks-cr/vars/vars.yml`, that sets three variables:

Variable	Value
<code>vsftpd_packages</code>	<code>vsftpd</code>
<code>vsftpd_service</code>	<code>vsftpd</code>
<code>vsftpd_config_file</code>	<code>/etc/vsftpd/vsftpd.conf</code>

- In your **ansible-vsftpd.yml** playbook, make sure that you use **vars\_files** to include the files of variables in the **ansible-playbooks-cr/vars** directory in your play.
- In the play in **ansible-vsftpd.yml**, create tasks which:
  1. Ensure that the packages listed by the variable `vsftpd_packages` are installed.
  2. Ensure that the services listed by the variable `vsftpd_service` are started and enabled to start at boot time.
  3. Use the **template** module to deploy the **templates/vsftpd.conf.j2** template to the location defined by the `vsftpd_config_file` variable. The file should be owned by user **root**, group **root**, have octal file permissions **0600**, and an SELinux type of **etc\_t**. Notify a handler that restarts **vsftpd** if this task cause a change.
  4. Ensure that the **firewalld** package is installed and that the service is started and enabled. Ensure that **firewalld** has been configured to immediately and permanently allow connections to the ftp service.
- In your **ansible-vsftpd.yml** playbook, create a handler to restart the services listed by the variable `vsftpd_service` when notified.

Create a third playbook, **site.yml**, in the **ansible-playbooks-cr** directory. This playbook should include the plays from the other two playbooks by containing exactly two lines:

```
- include: ansible-vsftpd.yml
- include: ftpclients.yml
```

You are encouraged to follow recommended playbook practices by naming all your plays and tasks. The playbooks should be written using appropriate modules, and should be able to be rerun safely. The playbooks should not make unnecessary changes to the systems.

Remember to use the **ansible-doc** command to help you find modules and information on how to use them.

When done, you should use **ansible-playbook site.yml** to check your work before running the grading script. You may also run the individual playbooks separately to make sure that they function.



## Important

If you are having trouble with your **site.yml** playbook, make sure that both **ansible-vsftpd.yml** and **ftpclients.yml** have indentation consistent with each other.

### Steps

1. As the **student** user on **workstation**, create the inventory file **/home/student/ansible-playbooks-cr/inventory/hosts**, containing **serverc.lab.example.com** in the group **ftpclients**, and **serverb.lab.example.com** and **serverd.lab.example.com** in the group **ftpservers**.
  - 1.1. Change directory into the Ansible project directory, **/home/student/ansible-playbooks-cr**, created by the setup script.

```
[student@workstation ~]$ cd /home/student/ansible-playbooks-cr
```

- 1.2. Create an inventory subdirectory, **inventory**.

```
[student@workstation ansible-playbooks-cr]$ mkdir inventory
```

- 1.3. Within the **/home/student/ansible-playbooks-cr/inventory** directory, create the static inventory file, **hosts**, by opening it with a text editor.

```
[student@workstation ansible-playbooks-cr]$ vim inventory/hosts
```

- 1.4. Populate the **hosts** file with the following contents:

```
[ftpservers]
serverb.lab.example.com
serverd.lab.example.com

[ftpclients]
serverc.lab.example.com
```

- 1.5. Save the changes to the newly created inventory file.

2. Create the Ansible configuration file, **/home/student/ansible-playbooks-cr/ansible.cfg**, and populate it with the necessary entries to configure the Ansible project to use the newly created inventory, connect to managed hosts as the **devops** user, and to utilize privilege escalation using **sudo** as the **root** user.

- 2.1. Create the Ansible configuration file, **/home/student/ansible-playbooks-cr/ansible.cfg**, by opening it with a text editor.

```
[student@workstation ansible-playbooks-cr]$ vim ansible.cfg
```

- 2.2. Configure the inventory, remote user, and privilege escalation method and user for the Ansible project by adding the following entries in the **ansible.cfg** configuration file.

```
[defaults]
remote_user = devops
inventory = ./inventory

[privilegeEscalation]
become_user = root
become_method = sudo
```

- 2.3. Save the changes to the newly created Ansible configuration file.

3. Create the playbook, **/home/student/ansible-playbooks-cr/ftpclient.yml**, containing a play targeting the hosts in the **ftpclients** inventory group and ensures that the **lftp** is installed.

- 3.1. Create the playbook file, **/home/student/ansible-playbooks-cr/ftpclient.yml**, by opening it with a text editor.

```
[student@workstation ansible-playbooks-cr]$ vim ftpclient.yml
```

- 3.2. Populate the new playbook file with a play to ensure that the *lftp* package is installed on the hosts in the **ftpclients** inventory group by adding the following entries.

```

- name: ftp client installed
 hosts: ftpclients
 become: true
 tasks:
 - name: latest lftp version installed
 yum:
 name: lftp
 state: latest
```

- 3.3. Save the changes to the newly created playbook file.

4. Create a **templates** subdirectory in the project working directory to hold the vsftpd configuration file provided.

- 4.1. Create the **templates** subdirectory.

```
[student@workstation ansible-playbooks-cr]$ mkdir templates
```

- 4.2. Download the provided **vsftpd.conf.j2** file to the newly created **templates** subdirectory.

```
[student@workstation ansible-playbooks-cr]$ curl -o templates/vsftpd.conf.j2
http://materials.example.com/ansible-playbooks-cr/templates/vsftpd.conf.j2
```

5. Create a **vars** subdirectory in the project working directory and populate it with the **defaults-template.yml** file provided.

- 5.1. Create the **vars** subdirectory.

```
[student@workstation ansible-playbooks-cr]$ mkdir vars
```

- 5.2. Download the provided **defaults-template.yml** file to the newly created **vars** subdirectory.

```
[student@workstation ansible-playbooks-cr]$ curl -o vars/defaults-template.yml
http://materials.example.com/ansible-playbooks-cr/vars/defaults-template.yml
```

6. Create a **vars.yml** variable definition file in the **vars** subdirectory to define the following three variables and their values.

Variable	Value
vsftpd_packages	vsftpd
vsftpd_service	vsftpd
vsftpd_config_file	/etc/vsftpd/vsftpd.conf

- 6.1. Create the **/home/student/ansible-playbooks-cr/vars/vars.yml** file.

```
[student@workstation ansible-playbooks-cr]$ vim vars/vars.yml
```

- 6.2. Populate the **vars.yml** file with the following variable definitions.

```
vsftpd_packages: vsftpd
vsftpd_service: vsftpd
vsftpd_config_file: /etc/vsftpd/vsftpd.conf
```

- 6.3. Save the changes to the newly created variable definition file.

7. Using the previously created Jinja2 template and variable definition files, create a second playbook, **/home/student/ansible-playbooks-cr/ansible-vsftpd.yml**, to configure the vsftpd service on the hosts in the **ftpservers** inventory group.

- 7.1. Create the playbook file, **/home/student/ansible-playbooks-cr/ansible-vsftpd.yml** by opening it with a text editor.

```
[student@workstation ansible-playbooks-cr]$ vim ansible-vsftpd.yml
```

- 7.2. Populate the new playbook file with the following entries in order to configure the vsftpd service on the hosts in the **ftpservers** inventory group.

```

- name: FTP server is installed
 hosts:
 - ftpservers

 become: true

 vars_files:
 - vars/defaults-template.yml
 - vars/vars.yml

 tasks:
 - name: Packages are installed
 yum:
 name: '{{ vsftpd_packages }}'
 state: installed

 - name: Ensure service is started
 service:
 name: '{{ item }}'
 state: started
 enabled: true
 with_items: '{{ vsftpd_service }}'
```

```
- name: Configuration file is installed
 template:
 src: templates/vsftpd.conf.j2
 dest: '{{ vsftpd_config_file }}'
 owner: root
 group: root
 mode: '0600'
 setype: etc_t
 notify: restart vsftpd

- name: firewalld is installed
 yum:
 name: firewalld
 state: present

- name: firewalld is started and enabled
 service:
 name: firewalld
 state: started
 enabled: yes

- name: Open ftp port in firewall
 firewalld:
 service: ftp
 permanent: true
 state: enabled
 immediate: yes

handlers:
 - name: restart vsftpd
 service:
 name: "{{ item }}"
 state: restarted
 with_items: "{{ vsftpd_service }}
```

- 7.3. Save the changes to the newly created playbook file.
8. Create a third playbook, **/home/student/ansible-playbooks-cr/site.yml**, and include the plays from the two playbooks created previously, **ftpclient.yml** and **ansible-vsftpd.yml**.
- 8.1. Create the playbook file, **/home/student/ansible-playbooks-cr/site.yml**, by opening it with a text editor.

```
[student@workstation ansible-playbooks-cr]$ vim site.yml
```

- 8.2. Populate the new playbook file with the following entries in order to include the plays from the other two playbooks.

```
Play for FTP clients
- include: ftpclient.yml

Play for FTP servers
- include: ansible-vsftpd.yml
```

- 8.3. Save the changes to the newly created playbook file.

9. Execute the **/home/student/ansible-playbooks-cr/site.yml** playbook to verify that it performs the desired tasks on the managed hosts.

```
[student@workstation ansible-playbooks-cr]$ ansible-playbook site.yml
```

#### Evaluation

As the **student** user on **workstation**, run the **lab ansible-playbooks-cr grade** command to confirm success of this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab ansible-playbooks-cr grade
```

#### Cleanup

Run the **lab ansible-playbooks-cr cleanup** command to clean up the lab tasks on **serverb**, **serverc**, and **serverd**.

```
[student@workstation ~]$ lab ansible-playbooks-cr cleanup
```

# Lab: Creating Roles and Using Dynamic Inventory

In this review, you will convert the **ansible-vsftpd.yml** playbook from the preceding exercise into a role, and then use that role in a new playbook that will also run some additional tasks. You will also be asked to install and use a dynamic inventory script, which will be provided to you.

## Outcomes

You should be able to:

- Create a role for configuring vsftpd by converting an existing playbook.
- Create and execute a playbook by including a role.
- Use dynamic inventory to execute a playbook.

## Before you begin

Log in to **workstation** as **student** using **student** as the password.

On **workstation**, run the **lab ansible-roles-cr setup** script. This script ensures that the host, **serverb.lab.example.com**, **serverc.lab.example.com**, and **serverd.lab.example.com** are reachable on the network. The script also checks that Ansible is installed on **workstation** and creates a directory structure for the lab environment and required Ansible configuration files.

```
[student@workstation ~]$ lab ansible-roles-cr setup
```

## Instructions

Start a new Ansible project by copying the files from the previous review's directory, **/home/student/ansible-playbooks-cr**, to the new project directory, **/home/student/ansible-roles-cr**, created by the setup script.

Download the dynamic inventory script from <http://materials.example.com/compreview/dynamic/crinventory.py>. Configure your Ansible project to use both the downloaded dynamic inventory script and the existing static inventory, **ansible-roles-cr/inventory/hosts**.

Convert **ansible-vsftpd.yml** into the role **ansible-vsftpd**, as specified below:

- Use **ansible-galaxy** to create the directory structure for the role **ansible-vsftpd** in the **ansible-roles-cr/roles** directory of your Ansible project.
- The file **ansible-roles-cr/vars/defaults-template.yml** contains default variables for the role that should be easy to override. It should be moved to an appropriate location in the role.
- The file **ansible-roles-cr/vars/vars.yml** contains regular variables for the role. It should be moved to an appropriate location in the role.
- The template **ansible-roles-cr/templates/vsftpd.conf.j2** should be moved to an appropriate location in the role.

- The tasks and handlers in the **ansible-vsftpd.yml** playbook should be appropriately installed in the role.
- You may edit the role's **meta/main.yml** file to set the author, description, and license fields (use BSD for the license). You may also edit the **README.md** file as you wish for completeness.
- Remove any directories in the role that you are not using.

Create a new playbook, **vsftpd-configure.yml**, in the **ansible-roles-cr** directory. It should be written as follows:

- It should contain a play targeting hosts in the inventory group **ftpservers**.
- The play should set the following variables:

Variable	Value
vsftpd_anon_root	/mnt/share
vsftpd_local_root	/mnt/share

- The play should apply the role **ansible-vsftpd**.
- The play should include the following tasks in the specified order:
  1. Use the **command** module to create a GPT disk label on **/dev/vdb**, that starts 1 MiB from the beginning of the device and ends at the end of the device. Use the **ansible-doc** command to learn how to use the **creates** argument to skip this task if **/dev/vdb1** has already been created. This is to avoid destructive repartitioning of the device. Use the following command to create the partition:

```
parted --script /dev/vdb mklabel gpt mkpart primary 1MiB 100%
```
  2. Ensure a **/mnt/share** directory exists for use as a mount point.
  3. Use **ansible-doc -l** to find a module that can make a file system on a block device. Use **ansible-doc** to learn how to use that module. Add a task to the playbook that uses it to create an XFS file system on **/dev/vdb1**. Do not force creation of that file system if one exists already.
  4. Add a task to ensure that **/etc/fstab** mounts the device **/dev/vdb1** on **/mnt/share** at boot, and that it is currently mounted. (Use **ansible-doc** to find a module that can help with this.) If this task changes, notify the **ansible-vsftpd** role's handler that restarts vsftpd.
  5. Add a task that ensures that the **/mnt/share** directory is owned by the **root** user and the **root** group, has the SELinux type defined in the **{{ vsftpd\_setype }}** variable from the role, and has octal permissions of **0755**. (This has to be done after the file system is mounted to set the permissions on the mounted file system and not on the placeholder mount point directory.)
  6. Make sure that a file named **README** exists in the directory specified by **{{ vsftpd\_anon\_root }}** containing the string "**Welcome to the FTP server at serverX.lab.example.com**" where **serverX.lab.example.com** is the actual fully-qualified hostname for that server. This file should have octal permissions of **0644**.

and the SELinux type specified by the `{} vsftpd_setype {}` variable. (Hint: look at the `copy` or `template` modules and the available Ansible facts in order to solve this problem.)



## Important

You may find it useful to debug your role by testing it in a playbook that does not contain the extra tasks or playbook variables listed above, but only contains a play that targets hosts in the group `ftpservers`, and applies the role.

Once you have confirmed that a simplified playbook using only the role works just like the original `ansible-vsftpd.yml` playbook, you can build the complete `vsftpd-configure.yml` playbook by adding the additional variables and tasks specified above.

Change the `ansible-roles-cr/site.yml` playbook to use the new `vsftpd-configure.yml` playbook instead of `ansible-vsftpd.yml`.

You are encouraged to follow recommended playbook practices by naming all your plays and tasks. The playbooks should be written using appropriate modules, and should be able to be rerun safely. The playbooks should not make unnecessary changes to the systems.

When done, use `ansible-playbook site.yml` to check your work before running the grading script. You may also run the individual playbooks separately to make sure they function.



## Important

If you are having trouble with your `site.yml` playbook, make sure that both `vsftpd-configure.yml` and `ftpclients.yml` have indentation consistent with each other.

## Evaluation

From `workstation`, run the `lab ansible-roles-cr grade` command to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab ansible-roles-cr grade
```

## Cleanup

Run the `lab ansible-roles-cr cleanup` command to clean up the lab tasks on `servera` and `serverb`.

```
[student@workstation ~]$ lab ansible-roles-cr cleanup
```

# Solution

In this review, you will convert the **ansible-vsftpd.yml** playbook from the preceding exercise into a role, and then use that role in a new playbook that will also run some additional tasks. You will also be asked to install and use a dynamic inventory script, which will be provided to you.

## Outcomes

You should be able to:

- Create a role for configuring vsftpd by converting an existing playbook.
- Create and execute a playbook by including a role.
- Use dynamic inventory to execute a playbook.

## Before you begin

Log in to **workstation** as **student** using **student** as the password.

On **workstation**, run the **lab ansible-roles-cr setup** script. This script ensures that the host, **serverb.lab.example.com**, **serverc.lab.example.com**, and **serverd.lab.example.com** are reachable on the network. The script also checks that Ansible is installed on **workstation** and creates a directory structure for the lab environment and required Ansible configuration files.

```
[student@workstation ~]$ lab ansible-roles-cr setup
```

## Instructions

Start a new Ansible project by copying the files from the previous review's directory, **/home/student/ansible-playbooks-cr**, to the new project directory, **/home/student/ansible-roles-cr**, created by the setup script.

Download the dynamic inventory script from <http://materials.example.com/comp-review/dynamic/crinventory.py>. Configure your Ansible project to use both the downloaded dynamic inventory script and the existing static inventory, **ansible-roles-cr/inventory/hosts**.

Convert **ansible-vsftpd.yml** into the role **ansible-vsftpd**, as specified below:

- Use **ansible-galaxy** to create the directory structure for the role **ansible-vsftpd** in the **ansible-roles-cr/roles** directory of your Ansible project.
- The file **ansible-roles-cr/vars/defaults-template.yml** contains default variables for the role that should be easy to override. It should be moved to an appropriate location in the role.
- The file **ansible-roles-cr/vars/vars.yml** contains regular variables for the role. It should be moved to an appropriate location in the role.
- The template **ansible-roles-cr/templates/vsftpd.conf.j2** should be moved to an appropriate location in the role.
- The tasks and handlers in the **ansible-vsftpd.yml** playbook should be appropriately installed in the role.

- You may edit the role's **meta/main.yml** file to set the author, description, and license fields (use BSD for the license). You may also edit the **README.md** file as you wish for completeness.
- Remove any directories in the role that you are not using.

Create a new playbook, **vsftpd-configure.yml**, in the **ansible-roles-cr** directory. It should be written as follows:

- It should contain a play targeting hosts in the inventory group **ftpservers**.
- The play should set the following variables:

Variable	Value
vsftpd_anon_root	/mnt/share
vsftpd_local_root	/mnt/share

- The play should apply the role **ansible-vsftpd**.
- The play should include the following tasks in the specified order:
  1. Use the **command** module to create a GPT disk label on **/dev/vdb**, that starts 1 MiB from the beginning of the device and ends at the end of the device. Use the **ansible-doc** command to learn how to use the **creates** argument to skip this task if **/dev/vdb1** has already been created. This is to avoid destructive repartitioning of the device. Use the following command to create the partition:

```
parted --script /dev/vdb mklabel gpt mkpart primary 1MiB 100%
```
  2. Ensure a **/mnt/share** directory exists for use as a mount point.
  3. Use **ansible-doc -l** to find a module that can make a file system on a block device. Use **ansible-doc** to learn how to use that module. Add a task to the playbook that uses it to create an XFS file system on **/dev/vdb1**. Do not force creation of that file system if one exists already.
  4. Add a task to ensure that **/etc/fstab** mounts the device **/dev/vdb1** on **/mnt/share** at boot, and that it is currently mounted. (Use **ansible-doc** to find a module that can help with this.) If this task changes, notify the **ansible-vsftpd** role's handler that restarts vsftpd.
  5. Add a task that ensures that the **/mnt/share** directory is owned by the **root** user and the **root** group, has the SELinux type defined in the **{{ vsftpd\_setype }}** variable from the role, and has octal permissions of **0755**. (This has to be done after the file system is mounted to set the permissions on the mounted file system and not on the placeholder mount point directory.)
  6. Make sure that a file named **README** exists in the directory specified by **{{ vsftpd\_anon\_root }}** containing the string "**Welcome to the FTP server at serverX.lab.example.com**" where **serverX.lab.example.com** is the actual fully-qualified hostname for that server. This file should have octal permissions of **0644** and the SELinux type specified by the **{{ vsftpd\_setype }}** variable. (Hint: look at the **copy** or **template** modules and the available Ansible facts in order to solve this problem.)



## Important

You may find it useful to debug your role by testing it in a playbook that does not contain the extra tasks or playbook variables listed above, but only contains a play that targets hosts in the group **ftpservers**, and applies the role.

Once you have confirmed that a simplified playbook using only the role works just like the original **ansible-vsftpd.yml** playbook, you can build the complete **vsftpd-configure.yml** playbook by adding the additional variables and tasks specified above.

Change the **ansible-roles-cr/site.yml** playbook to use the new **vsftpd-configure.yml** playbook instead of **ansible-vsftpd.yml**.

You are encouraged to follow recommended playbook practices by naming all your plays and tasks. The playbooks should be written using appropriate modules, and should be able to be rerun safely. The playbooks should not make unnecessary changes to the systems.

When done, use **ansible-playbook site.yml** to check your work before running the grading script. You may also run the individual playbooks separately to make sure they function.



## Important

If you are having trouble with your **site.yml** playbook, make sure that both **vsftpd-configure.yml** and **ftpclients.yml** have indentation consistent with each other.

### Steps

1. On **workstation**, populate the new Ansible project directory, **/home/student/ansible-roles-cr**, with the contents of the Ansible project directory from the previous review, **/home/student/ansible-playbooks-cr**.
  - 1.1. On **workstation** as the **student** user, change to the new Ansible project directory created by the setup script, **/home/student/ansible-roles-cr**.
 

```
[student@workstation ~]$ cd /home/student/ansible-roles-cr/
```
  - 1.2. Copy the contents of the previous review's Ansible project directory, **/home/student/ansible-playbooks-cr**, to the new Ansible project directory.
 

```
[student@workstation ansible-roles-cr]$ cp -r ../ansible-playbooks-cr/* .
```
2. Configure the new Ansible project to use both the dynamic inventory available at **http://materials.example.com/comp-review/dynamic/crinventory.py** as well as the static inventory, **ansible-roles-cr/inventory/hosts**.
  - 2.1. Download the provided dynamic inventory script to **inventory/crinventory.py**.
 

```
[student@workstation ansible-roles-cr]$ curl -o inventory/crinventory.py http://materials.example.com/comp-review/dynamic/crinventory.py
```

2.2. Make the dynamic inventory script executable.

```
[student@workstation ansible-roles-cr]$ chmod +x inventory/crinventory.py
```

3. Convert the **ansible-vsftpd.yml** playbook into the role **ansible-vsftpd**.

3.1. Using **ansible-galaxy**, create the directory structure for the new **ansible-vsftpd** role in the **roles** subdirectory of the Ansible project directory.

```
[student@workstation ansible-roles-cr]$ ansible-galaxy init --offline -p roles/ansible-vsftpd
- ansible-vsftpd was created successfully
```

3.2. Using **tree**, verify the directory structure created for the new role.

```
[student@workstation ansible-roles-cr]$ tree roles
roles
└── ansible-vsftpd
 ├── defaults
 │ └── main.yml
 ├── handlers
 │ └── main.yml
 ├── meta
 │ └── main.yml
 ├── README.md
 ├── tasks
 │ └── main.yml
 ├── tests
 │ └── inventory
 │ └── test.yml
 └── vars
 └── main.yml
```

3.3. Copy the variable definitions in the **vars/defaults-template.yml** to the **roles/ansible-vsftpd/defaults/main.yml** file.

```
[student@workstation ansible-roles-cr]$ cp vars/defaults-template.yml roles/ansible-vsftpd/defaults/main.yml
```

3.4. Copy the variable definitions in the **vars/vars.yml** to the **roles/ansible-vsftpd/vars/main.yml** file.

```
[student@workstation ansible-roles-cr]$ cp vars/vars.yml roles/ansible-vsftpd/vars/main.yml
```

3.5. Copy the **templates/vsftpd.conf.j2** to the **roles/ansible-vsftpd/templates/vsftpd.conf.j2** file.



## Important

Depending on the Ansible version you are using, you might encounter a situation where **ansible-galaxy** command does not create all the required subdirectories. For example, the **files** and **templates** directories might be missing. If this is the case, simply create those two directories manually using the **mkdir** command.

```
[student@workstation ansible-roles-cr]$ mkdir roles/ansible-vsftpd/templates; cp templates/vsftpd.conf.j2 roles/ansible-vsftpd/templates/
```

- 3.6. Copy the tasks in the **ansible-vsftpd.yml** playbook into the **roles/ansible-vsftpd/tasks/main.yml** file. The **roles/ansible-vsftpd/tasks/main.yml** file should have the following contents after you are done.

```

tasks file for ansible-vsftpd
- name: Packages are installed
 yum:
 name: '{{ vsftpd_packages }}'
 state: installed

- name: Ensure service is started
 service:
 name: '{{ item }}'
 state: started
 enabled: true
 with_items: '{{ vsftpd_service }}'

- name: Configuration file is installed
 template:
 src: vsftpd.conf.j2
 dest: '{{ vsftpd_config_file }}'
 owner: root
 group: root
 mode: '0600'
 setype: etc_t
 notify: restart vsftpd

- name: firewalld is installed
 yum:
 name: firewalld
 state: present

- name: firewalld is started and enabled
 service:
 name: firewalld
 state: started
 enabled: yes

- name: Open ftp port in firewall
 firewalld:
 service: ftp
 permanent: true
 state: enabled
 immediate: yes
```

- 3.7. Copy the handlers in the **ansible-vsftpd.yml** playbook into the **roles/ansible-vsftpd/handlers/main.yml** file. The **roles/ansible-vsftpd/handlers/main.yml** file should have the following contents after you are done.

```

handlers file for ansible-vsftpd
- name: restart vsftpd
 service:
 name: "{{ item }}"
 state: restarted
 with_items: "{{ vsftpd_service }}"
```

4. Modify the contents of the role's **meta/main.yml** file.

- 4.1. Change the value of the **author** entry to "**Red Hat Training**".

```
author: Red Hat Training
```

- 4.2. Change the value of the **description** entry to "**example role for D0407**".

```
description: example role for D0407
```

- 4.3. Change the value of the **company** entry to "**Red Hat**".

```
company: Red Hat
```

- 4.4. Change the value of the **license:** entry to "**BSD**".

```
license: BSD
```

5. Modify the contents of the role's **README.md** file so that it provides pertinent information regarding the role. After modification, the file should contain the following contents.

```
ansible-vsftpd
=====
```

Example ansible-vsftpd role from Red Hat's "Automation with Ansible" (D0407) course.

```
Requirements
```

-----

None.

```
Role Variables
```

-----

\* defaults/main.yml contains variables used to configure the vsftpd.conf template  
\* vars/main.yml contains the name of the vsftpd service, the name of the RPM package, and the location of the service's configuration file

```
Dependencies
```

-----

None.

#### Example Playbook

```

- hosts: servers
 roles:
 - ansible-vsftpd

License

BSD

Author Information

Red Hat (training@redhat.com)
```

6. Remove the unused directories from the new role.

```
[student@workstation ansible-roles-cr]$ rm -rf roles/ansible-vsftpd/tests
```

7. Create the new playbook **vsftpd-configure.yml**. It should contain the following contents.

```

- name: Install and configure vsftpd
 hosts: ftpservers
 become: true
 vars:
 vsftpd_anon_root: /mnt/share/
 vsftpd_local_root: /mnt/share/
 roles:
 - ansible-vsftpd
 tasks:
 - name: /dev/vdb1 is partitioned
 command: >
 creates=/dev/vdb1
 parted --script /dev/vdb mklabel gpt mkpart primary 1MiB 100%
 - name: XFS file system exists on /dev/vdb1
 filesystem:
 dev: /dev/vdb1
 fstype: xfs
 force: no
 - name: anon_root mount point exists
 file:
 path: '{{ vsftpd_anon_root }}'
 state: directory
 - name: /dev/vdb1 is mounted on anon_root
 mount:
 name: '{{ vsftpd_anon_root }}'
```

```

src: /dev/vdb1
fstype: xfs
state: mounted
dump: '1'
passno: '2'
notify: restart vsftpd

- name: Make sure permissions on mounted fs are correct
 file:
 path: '{{ vsftpd_anon_root }}'
 owner: root
 group: root
 mode: '0755'
 setype: "{{ vsftpd_setype }}"
 state: directory

- name: Copy README to the ftp anon_root
 copy:
 dest: '{{ vsftpd_anon_root }}/README'
 content: "Welcome to the FTP server at {{ ansible_fqdn }}\n"
 setype: '{{ vsftpd_setype }}'

```

8. Change the **site.yml** playbook to use the newly created **vsftpd-configure.yml** playbook instead of the **ansible-vsftpd.yml** playbook. The file should contain the following contents after you are done.

```

Play for FTP clients
- include: ftpclient.yml

Play for FTP servers
- include: vsftpd-configure.yml

```

9. Verify that the **site.yml** playbook works as intended by executing it with **ansible-playbook**.

```
[student@workstation ansible-roles-cr]$ ansible-playbook site.yml
```

#### Evaluation

From **workstation**, run the **lab ansible-roles-cr grade** command to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab ansible-roles-cr grade
```

#### Cleanup

Run the **lab ansible-roles-cr cleanup** command to clean up the lab tasks on **servera** and **serverb**.

```
[student@workstation ~]$ lab ansible-roles-cr cleanup
```

# Lab: Optimizing Ansible

In this review, you will deploy an updated web page to two web servers running behind a load balancer. You will use the **serial** keyword to push this update to one server at a time, and **delegate\_to** to remove web servers from the load balancer pool when being updated and to add them back again when the update completes.

The HAProxy load balancer is preconfigured on **serverc.lab.example.com**, and Apache HTTPD is preconfigured on **servera.lab.example.com** and **serverb.lab.example.com**.

After upgrading the web content, the web servers must be rebooted, one at a time so that site availability is unaffected, before adding them back to the load balancer pool.

## Outcomes

You should be able to:

- Delegate tasks to other hosts.
- Asynchronously run jobs in parallel.
- Use rolling updates.

## Before you begin

Log in to **workstation** as **student** using **student** as the password.

On **workstation**, run the **lab ansible-optimize-cr setup** script. It tests whether Ansible is installed on **workstation** and creates a directory structure for the lab environment with an inventory file. The script preconfigures **servera.lab.example.com** and **serverb.lab.example.com** as web servers and configures **serverc.lab.example.com** as the load balancer server using a round-robin algorithm. The script also creates a **templates** directory under the lab's working directory.

The inventory file **/home/student/ansible-optimize-cr/inventory/hosts** lists **servera.lab.example.com** and **serverb.lab.example.com** as managed hosts, which are members of the **[webservers]** group and also lists **serverc.lab.example.com** as part of the **[lbserver]** group.

```
[student@workstation ~]$ lab ansible-optimize-cr setup
```

## Instructions

Configure a playbook and supporting materials on **workstation** that meet the following criteria:

- As **student** on **workstation**, download the web page template located at **http://materials.example.com/jinja2/index-ver1.html.j2** to the **/home/student/ansible-optimize-cr/templates/** directory. It should have the following content:

```
<html>
<head><title>My Page</title></head>
<body>
<h1>
Welcome to {{ inventory_hostname }}.
```

```
</h1>
<h2>A new feature added.</h2>
</body>
</html>
```

- Create a playbook named **upgrade\_webserver.yml** in **/home/student/ansible-optimize-cr/**. The playbook should run on the host group, **webservers**. It should configure privilege escalation using the remote user, **devops**. It should use the **serial** method to push to one host at a time.

In addition, the playbook should execute the following tasks in the specified order:

1. Use the **haproxy** Ansible module to disable the web server in the load balancer pool named **app**. The host should be referred to using the **inventory\_hostname** variable. Use the socket, **/var/lib/haproxy/stats**. Wait until the server reports a status of **MAINT**. The task needs to be delegated to the server from the **[lbserver]** inventory group.
  2. Deploy the **index-ver1.html.j2** template to **/var/www/html/index.html**. Register the **pageupgrade** variable when this task runs.
  3. Reboot the server. Set an asynchronous delay of one second, do not poll, and ignore errors. Execute this task when **pageupgrade** changes.
  4. Use the **wait\_for** module to wait for the server to reboot. Determine this by waiting for the **sshd** (port 22) to open. Use the **inventory\_hostname** variable to determine which host to wait for. Delay 25 seconds before starting to poll, and time out after 200 seconds. Do not escalate privileges. Delegate this task to 127.0.0.1. Like the previous task, execute this task when **pageupgrade** changes.
  5. Use the **wait\_for** module to wait for the web server to be started. Use the **inventory\_hostname** variable to determine which host to wait for, and poll port 80. Time out after 20 seconds.
  6. Use the **haproxy** Ansible module to re-enable the web server in the load balancer pool named **app**. Use the same information as specified for the first task in this playbook.
- Run the completed playbook and manually confirm that everything worked. You can use the **curl** command to retrieve pages through the load balancer at **http://serverc.lab.example.com**.

Note that the template file you deployed customized the document root for the web site so that it will be clear which back-end server you are viewing through the load balancer on any given page reload.

#### Evaluation

From **workstation**, run the **lab ansible-optimize-cr** script with the **grade** argument to assess this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab ansible-optimize-cr grade
```

#### Cleanup

Run the **lab ansible-optimize-cr cleanup** command to clean up the lab tasks on **servera** and **serverb**.

---

```
[student@workstation ~]$ lab ansible-optimize-cr cleanup
```

## Solution

In this review, you will deploy an updated web page to two web servers running behind a load balancer. You will use the **serial** keyword to push this update to one server at a time, and **delegate\_to** to remove web servers from the load balancer pool when being updated and to add them back again when the update completes.

The HAProxy load balancer is preconfigured on **serverc.lab.example.com**, and Apache HTTPD is preconfigured on **servera.lab.example.com** and **serverb.lab.example.com**.

After upgrading the web content, the web servers must be rebooted, one at a time so that site availability is unaffected, before adding them back to the load balancer pool.

### Outcomes

You should be able to:

- Delegate tasks to other hosts.
- Asynchronously run jobs in parallel.
- Use rolling updates.

### Before you begin

Log in to **workstation** as **student** using **student** as the password.

On **workstation**, run the **lab ansible-optimize-cr setup** script. It tests whether Ansible is installed on **workstation** and creates a directory structure for the lab environment with an inventory file. The script preconfigures **servera.lab.example.com** and **serverb.lab.example.com** as web servers and configures **serverc.lab.example.com** as the load balancer server using a round-robin algorithm. The script also creates a **templates** directory under the lab's working directory.

The inventory file **/home/student/ansible-optimize-cr/inventory/hosts** lists **servera.lab.example.com** and **serverb.lab.example.com** as managed hosts, which are members of the **[webservers]** group and also lists **serverc.lab.example.com** as part of the **[lbserver]** group.

```
[student@workstation ~]$ lab ansible-optimize-cr setup
```

### Instructions

Configure a playbook and supporting materials on **workstation** that meet the following criteria:

- As **student** on **workstation**, download the web page template located at **http://materials.example.com/jinja2/index-ver1.html.j2** to the **/home/student/ansible-optimize-cr/templates/** directory. It should have the following content:

```
<html>
<head><title>My Page</title></head>
<body>
<h1>
Welcome to {{ inventory_hostname }}.
</h1>
<h2>A new feature added.</h2>
</body>
```

```
</html>
```

- Create a playbook named **upgrade\_webserver.yml** in **/home/student/ansible-optimize-cr/**. The playbook should run on the host group, **webservers**. It should configure privilege escalation using the remote user, **devops**. It should use the **serial** method to push to one host at a time.

In addition, the playbook should execute the following tasks in the specified order:

1. Use the **haproxy** Ansible module to disable the web server in the load balancer pool named **app**. The host should be referred to using the **inventory\_hostname** variable. Use the socket, **/var/lib/haproxy/stats**. Wait until the server reports a status of **MAINT**. The task needs to be delegated to the server from the **[lserver]** inventory group.
  2. Deploy the **index-ver1.html.j2** template to **/var/www/html/index.html**. Register the **pageupgrade** variable when this task runs.
  3. Reboot the server. Set an asynchronous delay of one second, do not poll, and ignore errors. Execute this task when **pageupgrade** changes.
  4. Use the **wait\_for** module to wait for the server to reboot. Determine this by waiting for the **sshd** (port 22) to open. Use the **inventory\_hostname** variable to determine which host to wait for. Delay 25 seconds before starting to poll, and time out after 200 seconds. Do not escalate privileges. Delegate this task to 127.0.0.1. Like the previous task, execute this task when **pageupgrade** changes.
  5. Use the **wait\_for** module to wait for the web server to be started. Use the **inventory\_hostname** variable to determine which host to wait for, and poll port 80. Time out after 20 seconds.
  6. Use the **haproxy** Ansible module to re-enable the web server in the load balancer pool named **app**. Use the same information as specified for the first task in this playbook.
- Run the completed playbook and manually confirm that everything worked. You can use the **curl** command to retrieve pages through the load balancer at **http://serverc.lab.example.com**.

Note that the template file you deployed customized the document root for the web site so that it will be clear which back-end server you are viewing through the load balancer on any given page reload.

### Steps

1. From **workstation** as the **student** user, change to the directory **/home/student/ansible-optimize-cr**.

```
[student@workstation ~]$ cd /home/student/ansible-optimize-cr
```

2. Because the web servers are preconfigured as part of the lab setup, use **curl** to browse **http://serverc.lab.example.com**. Run the **curl** command twice to see the web content from the web server running on **servera** and **serverb**.

```
[student@workstation ansible-optimize-cr]$ curl http://serverc.lab.example.com
```

```
<html>
<head><title>My Page</title></head>
<body>
<h1>
Welcome to servera.lab.example.com.
</h1>
</body>
</html>
```

```
[student@workstation ansible-optimize-cr]$ curl http://serverc.lab.example.com
<html>
<head><title>My Page</title></head>
<body>
<h1>
Welcome to serverb.lab.example.com.
</h1>
</body>
</html>
```

3. Create a new web page template named **index-ver1.html.j2** under the **templates** directory of the lab's working directory by downloading the web page template from <http://materials.example.com/jinja2/index-ver1.html.j2>.

```
[student@workstation ansible-optimize-cr]$ curl -o templates/index-ver1.html.j2
http://materials.example.com/jinja2/index-ver1.html.j2
```

4. Create a playbook named **upgrade\_webserver.yml** under `~/ansible-optimize-cr`. The playbook should use privilege escalation using the remote user **devops** and a **hosts** directive using the **webservers** host group.

The updates need to be pushed to one server at a time.

The contents of the **upgrade\_webserver.yml** file should be as follows:

```

- name: Upgrade Webservers
 hosts: webservers
 remote_user: devops
 become: yes
 serial: 1
```

5. Create a task in the **upgrade\_webserver.yml** playbook to remove the web server from the load balancer pool. Use the **haproxy** Ansible module to remove it from the HAProxy load balancer. The task needs to be delegated to a server from the **[lserver]** inventory group.

The **haproxy** module is used to disable a back-end server from HAProxy using socket commands. To disable a back-end server from the back-end pool named **app**, specify the socket path as **/var/lib/haproxy/stats**, and configure **wait=yes** so that the task waits until the server reports a status of **MAINT**.

The contents of the **upgrade\_webserver.yml** file should be as follows:

```
...file content omitted...
tasks:
```

```

- name: disable the server in haproxy
 haproxy:
 state: disabled
 backend: app
 host: "{{ inventory_hostname }}"
 socket: /var/lib/haproxy/stats
 wait: yes
 delegate_to: "{{ item }}"
 with_items: "{{ groups.labserver }}"

```

6. Create a task in the **upgrade\_webserver.yml** playbook to copy the updated page template from the lab working directory **templates/index-ver1.html.j2** to the web servers' document root directories as the file **/var/www/html/index.html**. Also register a variable, `pageupgrade`, which would be later used to invoke other tasks.

```

...file content omitted...
- name: upgrade the page
 template:
 src: "templates/index-ver1.html.j2"
 dest: "/var/www/html/index.html"
 register: pageupgrade

```

7. Create a task in the **upgrade\_webserver.yml** playbook to restart the web servers using an *asynchronous* task that will not wait more than **1** second for the task to complete, and that ensures tasks are not polled from completion.

Set `ignore_errors` to `true` and execute the task if the previously registered `pageupgrade` variable has changed.

- 7.1. Create a task in the **upgrade\_webserver.yml** playbook to reboot the web server by adding a task to the playbook. Use the **command** module to shut down the machine.

```

...file content omitted...
- name: restart machine
 shell: /bin/sleep 5 && shutdown -r now "Ansible updates triggered"

```

- 7.2. Continue editing the task in the **upgrade\_webserver.yml** playbook.

Use the **async** keyword to specify a 1 second wait time for task completion. Disable polling by setting the **poll** parameter to **0**. Set `ignore_errors` to `true` and execute the task if the earlier registered `pageupgrade` variable changes. Add the following lines in bold to the playbook:

```

...file content omitted...
- name: restart machine
 shell: /bin/sleep 5 && shutdown -r now "Ansible updates triggered"
 async: 1
 poll: 0
 ignore_errors: true
 when: pageupgrade.changed

```

8. Create a task in the **upgrade\_webserver.yml** playbook.

Delegate the task to **localhost**, and use the **wait\_for** module to wait for the server to be restarted. Specify the **host** as **inventory\_hostname**, **port** as **22**, **state** as

**started**, **delay** as **25**, and **timeout** as **200**. The task should be executed when the variable `pageupgrade` has changed. Privilege escalation is not required for this task.

The task in the `upgrade_webserver.yml` playbook should read as follows:

```
...file content omitted...
- name: wait for webserver to reboot
 wait_for:
 host: "{{ inventory_hostname }}"
 port: 22
 state: started
 delay: 25
 timeout: 200
 become: False
 delegate_to: 127.0.0.1
 when: pageupgrade.changed
```

9. Create a task in the `upgrade_webserver.yml` playbook to wait for the web server port to open. (The `httpd` service should already be enabled so it is started when the machine boots.) Specify the **host** as `inventory_hostname`, **port** as **80**, **state** as **started**, and **timeout** as **20**.

```
- name: wait for webserver to come up
 wait_for:
 host: "{{ inventory_hostname }}"
 port: 80
 state: started
 timeout: 20
```

10. Create a task in the `upgrade_webserver.yml` playbook to add the web server to the load balancer pool after the upgrade of the page. Use the `haproxy` Ansible module to add the server back to the HAProxy load balancer pool. The task needs to be delegated to the `serverc.lab.example.com` server, which is part of the `[lbserver]` inventory group.

The `haproxy` module is used to enable a back-end server from HAProxy using socket commands. To enable a back-end server from the **backend pool** named **app**, specify `/var/lib/haproxy/stats` for **socket path**, and set **wait** to **yes** so that the task waits for the server to report a status of healthy.

```
- name: enable the server in haproxy
 haproxy:
 state: enabled
 backend: app
 host: "{{ inventory_hostname }}"
 socket: /var/lib/haproxy/stats
 wait: yes
 delegate_to: "{{ item }}"
 with_items: "{{ groups.lbserver }}"
```

11. Review the contents of the playbook `upgrade-webserver.yml`.

```

- name: Upgrade Webservers
 hosts: webservers
 remote_user: devops
 become: yes
```

```

serial: 1

tasks:
 - name: disable the server in haproxy
 haproxy:
 state: disabled
 backend: app
 host: "{{ inventory_hostname }}"
 socket: /var/lib/haproxy/stats
 wait: yes
 delegate_to: "{{ item }}"
 with_items: "{{ groups.lbserver }}"

 - name: upgrade the page
 template:
 src: "templates/index-ver1.html.j2"
 dest: "/var/www/html/index.html"
 register: pageupgrade

 - name: restart machine
 shell: /bin/sleep 5 && shutdown -r now "Ansible updates triggered"
 async: 1
 poll: 0
 ignore_errors: true
 when: pageupgrade.changed

 - name: wait for webserver to reboot
 wait_for:
 host: "{{ inventory_hostname }}"
 port: 22
 state: started
 delay: 25
 timeout: 200
 become: False
 delegate_to: 127.0.0.1
 when: pageupgrade.changed

 - name: wait for webserver to come up
 wait_for:
 host: "{{ inventory_hostname }}"
 port: 80
 state: started
 timeout: 20

 - name: enable the server in haproxy
 haproxy:
 state: enabled
 backend: app
 host: "{{ inventory_hostname }}"
 socket: /var/lib/haproxy/stats
 wait: yes
 delegate_to: "{{ item }}"
 with_items: "{{ groups.lbserver }}"

```

12. Check the syntax of the playbook **upgrade-webserver.yml**. Resolve any syntax errors before proceeding to the next step.

You can compare your playbook to the one available for download from [http://materials.example.com/comp-review/playbooks/ansible-optimize/upgrade\\_webserver.yml](http://materials.example.com/comp-review/playbooks/ansible-optimize/upgrade_webserver.yml) or use the provided playbook in place of your own for the next step. Check the syntax using the **ansible-playbook --syntax-check** command.

```
[student@workstation ansible-optimize-cr]$ ansible-playbook --syntax-check
upgrade_webserver.yml
```

- Run the playbook **upgrade-webserver.yml** to upgrade the server.

The restart task takes several minutes, so move on to the next step when it reaches that point in executing the playbook.

```
[student@workstation ansible-optimize-cr]$ ansible-playbook upgrade_webserver.yml
```

- From **workstation**, use **curl** to view the web link **http://serverc.lab.example.com**. Run the **curl** command several times while the playbook is executing to verify the webserver is still reachable and that the host changes when the playbook moves on to reboot the other machine.
- Wait until the remaining tasks from the playbook complete on both **servera** and **serverb**.
- Verify the web content by browsing the website using the link **http://serverc.lab.example.com**. Rerun the **curl** command to see the updated pages from two different web servers, **servera.lab.example.com** and **serverb.lab.example.com**, with the updated content.

```
[student@workstation ansible-optimize-cr]$ curl http://serverc.lab.example.com
<html>
<head><title>My Page</title></head>
<body>
<h1>
Welcome to servera.lab.example.com.
</h1>
<h2>A new feature added.</h2>
</body>
</html>
```

```
[student@workstation ansible-optimize-cr]$ curl http://serverc.lab.example.com
<html>
<head><title>My Page</title></head>
<body>
<h1>
Welcome to serverb.lab.example.com.
</h1>
<h2>A new feature added.</h2>
</body>
</html>
```

### Evaluation

From **workstation**, run the **lab ansible-optimize-cr** script with the **grade** argument to assess this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab ansible-optimize-cr grade
```

**Cleanup**

Run the **lab ansible-optimize-cr cleanup** command to clean up the lab tasks on **servera** and **serverb**.

```
[student@workstation ~]$ lab ansible-optimize-cr cleanup
```

# Lab: Deploying Ansible Tower and Executing Jobs

In this review, you will deploy Ansible Tower on **tower.lab.example.com** and then launch a job using a job template.

## Outcome

You should be able to:

- Deploy Ansible Tower.
- Launch a job in Ansible Tower using a job template.

## Before you begin

Reset the **tower** system before beginning this lab.

Log in to **tower** as **root** using **redhat** as the password.

On **workstation**, run the **lab ansible-tower-cr setup** script. This script ensures that the tower host, **tower.lab.example.com**, is reachable on the network. The script also checks that Ansible is installed on **workstation**.

```
[student@workstation ~]$ lab ansible-tower-cr setup
```

## Instructions

Install Ansible Tower on **tower.lab.example.com**, and then launch a job using the job template **Demo Job Template**. Read through the entire list before you begin.

- Install Ansible Tower on **tower.lab.example.com** using the setup bundle **ansible-tower-setup-bundle.el7.tar.gz**, available at [http://content.example.com/ansible2.3/x86\\_64/dvd/ansible-tower/ansible-tower-setup-bundle-3.1.1-1.el7.tar.gz](http://content.example.com/ansible2.3/x86_64/dvd/ansible-tower/ansible-tower-setup-bundle-3.1.1-1.el7.tar.gz). A valid license file, **Ansible-Tower-license.txt**, is available at <http://materials.example.com/Ansible-Tower-license.txt>.

**Firefox** is available on **workstation** so that you can access Ansible Tower's web-based user interface once it has been installed.

The following parameters should be set in the **inventory** file used for installing Ansible Tower. Note that this will set the **admin** password for your Ansible Tower configuration to **redhat**.

## Ansible Tower configuration

Parameters	Values
<b>admin_password</b>	<b>redhat</b>
<b>pg_password</b>	<b>redhat</b>
<b>rabbitmq_password</b>	<b>redhat</b>

- Once Ansible Tower has been successfully installed, as the **student** user on **workstation**, change directory to the **/home/student/ansible-tower-cr** Ansible project directory

---

created by the setup script, and execute the `mkdemoproject.yml` playbook against the `tower.lab.example.com` managed host using the following command.

```
[student@workstation ~]$ cd /home/student/ansible-tower-cr; ansible-playbook
mkdemoproject.yml
```

- Log in to Ansible Tower as the `admin` user and launch a job using the job template **Demo Job Template**.

#### Evaluation

In the Tower web interface, determine the outcome of the job execution. When the job has completed successfully, the `STATUS` value changes to **Successful**. Review the output of the job execution. You should see that the `msg` module was used to successfully display a "**Hello World!**" message.

## Solution

In this review, you will deploy Ansible Tower on **tower.lab.example.com** and then launch a job using a job template.

### Outcome

You should be able to:

- Deploy Ansible Tower.
- Launch a job in Ansible Tower using a job template.

### Before you begin

Reset the **tower** system before beginning this lab.

Log in to **tower** as **root** using **redhat** as the password.

On **workstation**, run the **lab ansible-tower-cr setup** script. This script ensures that the tower host, **tower.lab.example.com**, is reachable on the network. The script also checks that Ansible is installed on **workstation**.

```
[student@workstation ~]$ lab ansible-tower-cr setup
```

### Instructions

Install Ansible Tower on **tower.lab.example.com**, and then launch a job using the job template **Demo Job Template**. Read through the entire list before you begin.

- Install Ansible Tower on **tower.lab.example.com** using the setup bundle **ansible-tower-setup-bundle.el7.tar.gz**, available at [http://content.example.com/ansible2.3/x86\\_64/dvd/ansible-tower/ansible-tower-setup-bundle-3.1.1-1.el7.tar.gz](http://content.example.com/ansible2.3/x86_64/dvd/ansible-tower/ansible-tower-setup-bundle-3.1.1-1.el7.tar.gz). A valid license file, **Ansible-Tower-license.txt**, is available at <http://materials.example.com/Ansible-Tower-license.txt>.

**Firefox** is available on **workstation** so that you can access Ansible Tower's web-based user interface once it has been installed.

The following parameters should be set in the **inventory** file used for installing Ansible Tower. Note that this will set the **admin** password for your Ansible Tower configuration to **redhat**.

### Ansible Tower configuration

Parameters	Values
<b>admin_password</b>	<b>redhat</b>
<b>pg_password</b>	<b>redhat</b>
<b>rabbitmq_password</b>	<b>redhat</b>

- Once Ansible Tower has been successfully installed, as the **student** user on **workstation**, change directory to the **/home/student/ansible-tower-cr** Ansible project directory created by the setup script, and execute the **mkdemoproject.yml** playbook against the **tower.lab.example.com** managed host using the following command.

```
[student@workstation ~]$ cd /home/student/ansible-tower-cr; ansible-playbook mkdemoproject.yml
```

- Log in to Ansible Tower as the **admin** user and launch a job using the job template **Demo Job Template**.

#### Steps

1. On **tower**, as the **root** user, change to the directory, and download the Ansible Tower setup bundle located at [http://content.example.com/ansible2.3/x86\\_64/dvd/ansible-tower/ansible-tower-setup-bundle-3.1.1-1.el7.tar.gz](http://content.example.com/ansible2.3/x86_64/dvd/ansible-tower/ansible-tower-setup-bundle-3.1.1-1.el7.tar.gz).

```
[root@tower ~]# curl -O -J http://content.example.com/ansible2.3/x86_64/dvd/ansible-tower/ansible-tower-setup-bundle-3.1.1-1.el7.tar.gz
```

2. Extract the setup bundle, **ansible-tower-setup-bundle-3.1.1-1.el7.tar.gz**.

```
[root@tower ~]# tar xzf ansible-tower-setup-bundle-3.1.1-1.el7.tar.gz
```

3. Change into the directory containing the extracted contents.

```
[root@tower ~]# cd ansible-tower-setup-bundle-3.1.1-1.el7
```

4. Set the passwords for the Ansible Tower administrator account, database user account, and messaging user account to **redhat**. Do this by modifying their respective entries in the **inventory** file used by the Tower installer playbook.

```
[root@tower ansible-tower-setup-bundle-3.1.1-1.el7]# grep password inventory
admin_password='redhat'
pg_password='redhat'
rabbitmq_password='redhat'
```

5. Run the Ansible Tower installer by executing the **setup.sh** script. The script may take up to 30 minutes to complete. Ignore the errors in the script output. They are related to verification checks performed by the installer playbook.

```
[root@tower ansible-tower-setup-bundle-3.1.1-1.el7]# ./setup.sh
[warn] Will install bundled Ansible
Loaded plugins: langpacks, search-disabled-repos
Examining bundle/repos/epel/ansible-2.2.1.0-1.el7.noarch.rpm:
 ansible-2.2.1.0-1.el7.noarch
Marking bundle/repos/epel/ansible-2.2.1.0-1.el7.noarch.rpm to be installed
... Output omitted ...
The setup process completed successfully.
Setup log saved to /var/log/tower/setup-2017-02-27-10:52:44.log
```

6. Once the installer has completed successfully, exit the console session on the **tower** system.

```
[root@tower ansible-tower-setup-bundle-3.1.1-1.el7]# exit
```

7. Launch the Firefox web browser from **workstation** and connect to your Ansible Tower at <https://tower.lab.example.com>. Firefox warns you that the Ansible Tower server's security certificate is not secure. Add and confirm the security exception for the self-signed certificate.
8. Log in to the Tower web interface as the Tower administrator using the **admin** account and the **redhat** password.
9. Once you have successfully logged in to the Tower web interface for the first time, you are prompted to enter a license and accept the end user license agreement.  
Upload the Ansible Tower license and accept the end user license agreement.
  - 9.1. On **workstation**, download the Ansible Tower license provided at <http://materials.example.com/Ansible-Tower-license.txt>.
  - 9.2. In the Tower web interface, click **BROWSE** and then select the license file downloaded earlier.
  - 9.3. Select the checkbox next to **I agree to the End User License Agreement** to indicate acceptance.
  - 9.4. Click **SUBMIT** to submit the license and accept the license agreement.
10. Once Ansible Tower has been successfully installed, as the **student** user on **workstation**, change directory to the **/home/student/ansible-tower-cr** Ansible project directory created by the setup script and execute the **mkdemoproject.yml** playbook against the **tower.lab.example.com** managed host using the following command. This installs a demo job template on the Tower server.

```
[student@workstation ~]$ cd /home/student/ansible-tower-cr; ansible-playbook mkdemoproject.yml
```

11. In the Tower web interface, identify the Job Template created during the Ansible Tower installation. Click **TEMPLATES** in the top navigation menu to display the list of existing Job Templates. You should see a Job Template named **Demo Job Template**, created during the Ansible Tower installation.
12. Launch a job using the **Demo Job Template** Job Template.
  - 12.1. Exit the template details view by clicking the **TEMPLATES** link in the breadcrumb navigation menu near the top of the screen.
  - 12.2. On the **TEMPLATES** screen, click the rocket icon under the **ACTIONS** column of the **Demo Job Template** row. This launches a job using the parameters configured in the **Demo Job Template** template and redirects you to the job details screen. As the job executes, the details of the job execution, as well as its output, is displayed.

### Evaluation

In the Tower web interface, determine the outcome of the job execution. When the job has completed successfully, the **STATUS** value changes to **Successful**. Review the output of the

job execution. You should see that the **msg** module was used to successfully display a "**Hello World!**" message.





## **APPENDIX A**

# **ANSIBLE LIGHTBULB LICENSING**

## Ansible Lightbulb License

Portions of this course were adapted from the Ansible Lightbulb project. The original material from that project is available from <https://github.com/ansible/lightbulb> under the following MIT License:

Copyright 2017 Red Hat, Inc.

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.